Pontifícia Universidade Católica do Rio de Janeiro

Diego Cedrim Gomes Rêgo

# Understanding and Improving Batch Refactoring in Software Systems

Tese de Doutorado

Thesis presented to the Programa de Pós–Graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática.

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
September 2018

Pontifícia Universidade Católica do Rio de Janeiro

Diego Cedrim Gomes Rêgo

# Understanding and Improving Batch Refactoring in Software Systems

Thesis presented to the Programa de Pós–Graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática. Approved by the undersigned Examination Committee.

Rio de Janeiro, September 28th, 2018

**Diego Cedrim Gomes Rêgo**

The author received his Bachelor degree in Computer Science from the Instituto de Computação (IC) of Universidade Federal de Alagoas (UFAL) in 2007. He also received his Master degree in Computer Science from Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio) in 2009. During his academic career, he participated in several research projects. His main research interests are: Code Smells, Refactoring, Software Architecture, and Empirical Software Engineering.

Bibliographic data

Rêgo, Diego Cedrim Gomes

Understanding and Improving Batch Refactoring in Software Systems / Diego Cedrim Gomes Rêgo ; advisor: Alessandro Fabricio Garcia. – 2018.
168 f. : il. color. ; 30 cm

Tese (doutorado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2018.
Inclui bibliografia

1. Informática – Teses. 2. Refatoração. 3. Anomalias de código. 4. Manutenção de software. 5. Evolução de software. 6. Qualidade estrutural. I. Garcia, Alessandro Fabricio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my family, for their support
and encouragement.

## Acknowledgments

I have to start by thanking my wonderful family. I'm absolutely sure that, without them, I would not have accomplished anything minimally significant in my life. My mom, Betânia Cedrim, is my true hero. She is the strongest person I know and she is astonishingly amazing! I will be eternally grateful to her. I love you, mom! I also thank my father, Murilo Rêgo Júnior for, being always by my side. Without him, I would never be a good student. He was my main driving force to keep studying hard. Thank you, dad. I love you! Words cannot express my love to my brother Tiago Cedrim. I'm so lucky for having him in my life. Thank you for all the support, encouragement, and the long conversations about our lives. You always were there to give me strength in the hardest times. You truly inspire me. I also want to thank my sister-in-law, Bruna Spencer, for making him so happy. Love you all! I would like to also thank my cousin Bruno Martins. He truly is my second brother. Thank you for all support, care, and love throughout so many years. I love you!

I also want to thank my beloved grandparents Bernadete Cedrim and Deraldo Cedrim. They helped my mom to raise me with so much love and care. Thank you for such amazing conversations and guidance. For sure, you are wonderful persons. I must thank my uncle Deraldo Cedrim Júnior. He is like my third dad and always helped and inspired me in so many ways. He was there in the hardest moment and provided me with a good education. I owe you all my accomplishments, my beloved uncle.

I thank my beloved wife, Juliana Leal. She is with me for nine wonderful years filling my life with joy and love. She always supported me during this Ph.D., even when this meant a long-distance relationship for quite a while. Thank you so much for your support and encouragement. Always count on me. I truly love you!

My deepest gratitude to my advisor Alessandro Garcia. Without his guidance and hardworking, I would not have done anything valuable during this Ph.D. Thank you for all the opportunities and for the amazing experiences you provided me. Thanks to you, I could make several of my dreams to come true during these 4.5 years. I will be eternally grateful to you. Always count on me!

During this Ph.D., I was able to meet incredible people who I'm proud to say that are my friends. Thank you to Leonardo Sousa, who helped me so much in so many opportunities. You are a true inspiration for the people around you, be aware of that. Thank you for giving me the gift of your friendship, which I know is for life. I consider you my brother and you can always count on me. I also want to thank Roberto Oliveira, who is the third member of the “three musketeers.” Roberto is an amazing person. You and your history inspire me so much. Thank you for all the amazing moments we shared these years. Last but not least, I want to thank Anderson Oliveira for his friendship and permanent willingness to help.

I also thank all the professors from DI. Their contribution to my education is invaluable. I owe them all my accomplishments so far and the ones to come. I also want to thank the members of my thesis defense team: Simone Barbosa, Marcos Kalinowski, Guilherme Travassos, and Rohit Gheyi, which come from so far to evaluate my work. I also want to thank Rohit for all the incredible feed-backs and contributions during my Ph.D. Thank you!

This study was financied in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. I also want to thank CNPq, FAPERJ, and PUC-Rio for the financial support that made my research possible in the first place. Finally, my sincere thanks to the administrative staff of the DI at PUC-Rio.

## Abstract

Rêgo, Diego Cedrim Gomes; Garcia, Alessandro (Advisor). **Understanding and Improving Batch Refactoring in Software Systems**. Rio de Janeiro, 2018. 168p. Tese de doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Code smells in a program represent indications of structural quality problems, which can be addressed by software refactoring. However, developers may neglect or end up creating new code smells through single refactoring. Little has been reported about recurring beneficial and harmful effects of refactoring on the program structural quality. As a consequence, developers still miss guidance along non-trivial smell-removing tasks. In fact, evidence suggests developers often need to apply a sequence of refactorings, so-called batch refactoring, to entirely remove a smelly code structure. Thus, in this thesis, we have conducted a series of studies to understand the impact of single and batch refactorings on code smells. In our first studies, we analyze how often commonly-used types of single refactoring affect the density of code smells along the version histories of dozens of projects. Even though 79.4% of the refactorings touched smelly elements, 57% had no impact on the smell removal. Surprisingly, only 9.7% of refactorings removed smells, while 33% induced the introduction of new ones. On one hand, we observed that harmful refactoring-smell patterns could be used to guide developers to avoid smell-inducing refactoring. On the other hand, we observed that many smells can be removed only through batch refactoring. Thus, our last studies investigate the impact of batch refactorings on smells. Even when applied in batches, refactorings tend to maintain or even increase the density of code smells. We also identified common batch-smell patterns, which enable us to create heuristics that can guide developers through smell-removing tasks. The last study evaluated those heuristics, and we conclude the outcomes are promising.

## Keywords

Refactoring; Code Smells; Software Maintenance; Software Evolution; Structural Quality.

## Resumo

Rêgo, Diego Cedrim Gomes; Garcia, Alessandro. **Entendendo e Melhorando a Prática de Refatorações em Lote em Sistemas de Software**. Rio de Janeiro, 2018. 168p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Em um sistema de software, as anomalias de código indicam problemas estruturais que podem ser resolvidos através da refatoração. No entanto, desenvolvedores podem negligenciar ou acabar criando novas anomalias ao refatorar. Pouco foi relatado sobre os efeitos benéficos e prejudiciais da refatoração de anomalias de código. Evidências sugerem que os desenvolvedores frequentemente precisam aplicar uma sequência de refatorações (refatoração em lote) para remover completamente as estruturas anômalas. Assim, nesta tese, realizamos uma série de estudos para entender o impacto de refatorações simples e em lote em anomalias de código. Em nossos primeiros estudos, analisamos com que frequência os tipos de refatoração comumente usados afetam a densidade de anomalias ao longo das histórias de dezenas de projetos. Mesmo que 79,4% das refatorações tenham tocado em elementos anômalos, 57% não reduziram suas ocorrências. Surpreendentemente, apenas 9,7% das refatorações removeram anomalias de código, enquanto 33% induziram a introdução de novas. Por um lado, observamos padrões nocivos de introdução de anomalias. Por outro lado, observamos que muitas anomalias podem ser removidas apenas por refatorações em lote. Assim, nossos últimos estudos investigam o impacto de refatorações em lote nas anomalias. Mesmo quando aplicadas em lotes, as refatorações tendem a não afetar ou mesmo aumentar a densidade de anomalias. Também identificamos padrões entre tipos de lotes e tipos de anomalias, levando-nos à criação de heurísticas que podem orientar os desenvolvedores durante tarefas de remoção de anomalias de código. O último estudo avaliou essas heurísticas e concluímos que os resultados são promissores.

## Palavras-chave

Refatoração; Anomalias de Código; Manutenção de Software; Evolução de Software; Qualidade Estrutural.

# Table of contents

# List of figures

# List of tables

*I've got to keep running the course,*
*I've got to keep running and win at all costs,*
*I've got to keep going, be strong,*
*Must be so determined and push myself on.*

**Iron Maiden**, *The Loneliness Of The Long Distance Runner.*

# 1
# Introduction

Developers often need to improve the software structural quality in order to satisfy evolving requirements. For such purpose, developers apply a common practice along the software maintenance, known as refactoring [1, 2]. Refactoring is a program transformation used for improving the structure of a program while preserving its observable behavior [3]. Examples of commonly applied types of refactoring include [1]: (i) restructuring or moving class members, such as *Extract Method*, *Move Method* and *Pull Up Method*, and (ii) extracting new elements, such as *Extract Superclass* and *Extract Interface*. The refactoring process comprises four steps [4]: (i) identification of program structures that should be refactored; (ii) appropriate selection of which refactoring types should be applied; (iii) refactoring implementation, and (iv) understanding if the applied refactoring actually improved the program structure.

The identification of program structures that should be refactored is particularly difficult for large projects. To facilitate the identification of such structures, Fowler [3] proposed the notion of code smells. A code smell represents an indication of software structural problems in a program [3], in which can be used as a hint for refactoring. In this way, each refactoring can contribute to removing at least one code smell. Examples of common smell types include *God Class*, *Long Method* and *Speculative Generality* [5, 6, 7]. Recent studies conclude that several types of code smells are recognized as critical by software developers [6, 7]. Since each smell can be a hint for one or more refactorings, one might expect that the smell used as hint will be removed after the refactoring has been applied. However, such expectation may not hold in practice. In fact, developers may neglect or even introduce other smells while refactoring. Unfortunately, the accumulate introduction of code smells can create more severe structural problems, which can lead to the system architectural degradation [5].

Studies suggest that refactoring is often used to improve the software structural quality [1, 8]. This improvement can be achieved through two refactoring tactics, namely *root-canal refactoring* and *floss refactoring*. Root-canal refactoring is used for strictly improving the source code structure and consists of pure refactoring, *i.e.*, refactoring is not performed in conjunction with other non-refactoring changes. Floss refactoring consists of refactoring the code together with non-structural changes as a means to reach other goals, such as adding features or removing bugs. Floss refactoring interleaves structural improvement with other programming activities and it is the most common tactic applied [1]. Since developers do several code changes during floss refactoring, they may often end up introducing, rather than reducing, code smells. Moreover, even when developers perform root-canal refactoring, they may fail to remove a smell or they may introduce new ones. Independently of refactoring tactic employed, the increase of code smells in a program is harmful. The increased density of code smells may not only induce design degradation [5, 9, 10], but also increase fault proneness [11, 12], and future maintenance effort [13, 14].

Researchers have been trying to understand how and why developers perform refactoring [1, 15, 16]. For instance, these studies report the most common refactoring types applied by developers [1], the typical reasons underlying several refactoring types [15], and how developers use tools that help to refactor [16]. Unfortunately, these studies have not addressed whether and how often developers successfully remove code smells during refactoring. In this vein, Bavota *et al.* [17] performed a study aiming to investigate if refactoring tends to remove code smells in real projects settings. However, they studied a small sample of three projects, making the results not necessarily generalizable. Besides that, they did not investigate cases of refactoring introducing code smells.

The relation between refactoring and smells is likely to be more complex than usually assumed by existing research [17]. For instance, to remove a single smell, a developer may need to apply a sequence of single refactoring, which is named *batch refactoring* in the literature [1]. Previous studies have not investigated if developers introduce or remove smells when applying either single refactoring operations or batch refactoring.

## 1.1 Refactoring Smelly Code Elements: A Motivating Example

A code smell is a surface indication that usually corresponds to a deeper problem in the system [3]. When a code element is affected by at least one code smell, we call it a *smelly element*. Since code smells can be hints to apply refactoring, one might expect that smelly elements become free from smells after refactoring. Conversely, elements may eventually become smelly after refactoring. Various reasons may lead to this problem, including the wrong selection of the refactoring type or lack of attention from the developer. To illustrate both scenarios, let us consider the following example, which refers to a customer management system. This system is responsible for storing and managing customer data of a particular company. Among its classes, the system contains the *Phone* and *Customer* classes, which are presented below:

```java
public class Phone {
    private final String unformattedNumber;
    public Phone(String unformattedNumber) {
        this.unformattedNumber = unformattedNumber;
    }
    public String getAreaCode() {
        return unformattedNumber.substring(0,3);
    }
    public String getPrefix() {
        return unformattedNumber.substring(3,6);
    }
    public String getNumber() {
        return unformattedNumber.substring(6,10);
    }
}

public class Customer {
    private Phone mobilePhone;
    public String getMobilePhoneNumber() {
        return "(" +
        mobilePhone.getAreaCode() + ") " +
        mobilePhone.getPrefix() + "-" +
        mobilePhone.getNumber();
    }
}
```

**Refactoring and Removing a Code Smell.** The method *getMobilePhoneNumber* of *Customer* class is an example of a smelly element. This method is affected by the code smell known as *Feature Envy*, which refers to a method that is more interested in other class than the one to which it belongs. In this example, *getMobilePhoneNumber* calls more methods belonging to the *Phone* class than those declared in its own class. Hence, the method seems to be more interested in the *Phone* class than in the class *Customer*, which leads to the *Feature Envy*. Since this method only calls methods from the *Phone* class, it should be declared in there. A developer can apply the *Move Method* refactoring to move the "envious method" from its current class to the class to which it is interested. Thus, the code smell served as a hint for the refactoring is likely to be removed from the source code after refactoring, thereby improving the program structure. The result of applying the *Move Method* refactoring can be seen below.

```java
public class Phone {
    private final String unformattedNumber;
    public Phone(String unformattedNumber) {
        this.unformattedNumber = unformattedNumber;
    }
    public String getAreaCode() {
        return unformattedNumber.substring(0,3);
    }
    public String getPrefix() {
        return unformattedNumber.substring(3,6);
    }
    public String getNumber() {
        return unformattedNumber.substring(6,10);
    }
    public String toFormattedString() {
        return "(" + this.getAreaCode() + ") " + this.
           getPrefix() + "-" + this.getNumber();
    }
}

public class Customer {
    private Phone mobilePhone;
    public Phone getPhone() {
        return this.mobilePhone;
    }
    //additional omitted methods
}
```

In this example, the code smell was used as a hint for refactoring as well as an indicator of which refactoring type the developer should apply. However, notice that the *Move Method* was not the only refactoring applied. The developer had to perform other changes to complete the *Move Method* refactoring. After moving the method, he updated all references to the *mobilePhone* object to reference the current object through the keyword *this*. Then, he applied the *Rename Method* refactoring to change the method's name from *getMobilePhoneNumber* to *toFormattedString*. He had to apply such refactoring because the old name is not suitable anymore. Additionally, the developer created a new method on the *Customer* class to make the private object *mobilePhone* externally accessible. After all these changes, the new *toFormattedString* method only calls methods from the class to which it belongs. Consequently, the developer removed the *Feature Envy* code smell by applying a *Move Method* refactoring.

**Smell-Inducing Refactoring.** The above example shows a scenario where the developer applied a refactoring that had a positive influence in the code structure since it got rid of an existing code smell. Unfortunately, this is not always the case. For instance, let us suppose the developer still wants to improve the system structural quality. For instance, he realized that the *Customer* class represents only a specific category of people. The current program structure prevents to introduce other categories of people, such as *Employee* and *Contractor*, to be added in the future. To address this matter, he decided to create a new abstraction (superclass) to represent *Person*. He also moved some of the members of the *Customer* class to the new *Person* class. Thus, other entities that also contain a phone number can benefit from the new abstraction. The *Person* superclass can be used later to add new entities to the system. In order to execute this generalization process, he applied an *Extract Superclass* refactoring. In this refactoring, the developer creates a new superclass so that its future subclasses can inherit data and behavior from it. The structure of the resulting classes can be seen below.

```java
public class Person {
    private Phone mobilePhone;
    public Phone getPhone() {
        return this.mobilePhone;
    }
}
public class Customer extends Person {
    //additional omitted methods
}
```

After applying the *Extract Superclass* refactoring, the new superclass is extended by the existing ones. Before applying the refactoring, the developer speculated about the introduction of new classes in the system, such as *Employee* and *Contractor*. All of them would also extend the new *Person* class. However, these classes have never been added to the system, and the existing classes continued to use the *Customer* class. Consequently, by applying the *Extract Superclass* refactoring, the developer introduced complexity in the system by creating an unused hierarchy to support anticipated future features that never have been implemented. This unused hierarchy represents a code smell named *Speculative Generality*. Therefore, in this case, the developer introduced a new code smell while refactoring, and ended up not removing it afterwards.

## 1.2 Problem Statement and Limitations of Related Work

In the previous section, we illustrated scenarios in which refactoring are applied to a program. We also presented an example that illustrates how developers may degrade the code structure rather than improving it while refactoring. In the example, the developer ended up introducing a new code smell after using the root-canal refactoring tactic. Hence, refactoring can be harmful even when this pure-refactoring tactic is used. Unfortunately, little has been reported about whether and to what extent developers introduce or remove code smells through refactoring. This lack of knowledge might let the developers unaware of the risks of introducing structural problems while refactoring.

Even though studies have been investigating the effects of refactoring on software systems [11, 13, 17, 18], most of them only address the relation between smells and refactoring superficially. For instance, Bavota *et al.* [17] performed the first study aiming to investigate if refactoring removes code smells in real project settings. They studied refactoring operations from two perspectives: (i) how often they are performed on classes exhibiting code smells, and (ii) whether they were able to remove smells or not. However, their study was limited in several ways. First, they have not explored scenarios where refactorings introduced new code smells. Second, they analyzed only three systems. Third, they only considered systems' major versions during refactoring collection. Consequently, the decision to look at major versions can lead to significant differences in the collected refactorings since they can miss refactoring operations that happened between minor versions, thus, hampering one's confidence on their findings.

Despite Bavota *et al.*'s first attempt to understand the relation between refactoring and smells, they fell short of revealing to what extent refactorings decrease, rather than increase, the density of smells in software projects. Thus, we still do not know if and how often refactoring introduces or removes smells. Therefore, in order to derive this knowledge, we need to investigate whether refactorings interfere either positively or negatively on the presence of smells (Research Problem 1).

**Research Problem 1.** Whether refactoring interferes on the density of code smells is unknown.

By addressing the first research problem, we can reveal if and how often refactorings remove or introduce code smells. However, the outcome of this analysis does not suffice to derive insights about the relationship between smells and specific refactoring types. As aforementioned, this relation is often complex. Thus, a follow-up investigation is to check when specific refactoring types tend to introduce, neglect or remove specific smell types. An existing catalog of code refactorings [3] establishes some relations between refactoring and smell types. For instance, an *Extract Method* refactoring is expected to remove a *Long Method* smell. However, there is no empirical knowledge on: (i) how often such refactoring-smell relations are observed in practice, and (ii) which other common refactoring-smell relations are not covered by the catalog. This follow-up investigation is important because we might be able to discover unknown relationships between types of refactorings and smells. The latter could be used to better guide developers. For instance, let us assume that we observe that when a particular refactoring type $r_t$ is applied, very often a specific code smell $s$ is introduced. This knowledge might be useful to increase the developer's awareness about a potentially harmful effect of applying a new refactoring of the type $r_t$. This investigation can be used to address the following research problem.

**Research Problem 2.** When refactoring introduces, neglects or removes smells is unknown.

By addressing these two research problems, tool designers can explore the derived knowledge to create alerts about smell-inducing refactorings that should be avoided. Such a tool could also suggest additional refactorings. For instance, suppose that a developer applied a single refactoring, but he failed to remove the smell completely. The tool could suggest a second refactoring to assist him to fully remove the smell.

In fact, developers may need to apply a sequence of refactorings to remove a smell. In practice, 40% of the times developers apply two or more refactorings in the same code element [1]. We call batch refactoring when developers apply a sequence of refactorings. Thus, we should not ignore batch refactorings if we want to understand the relation between refactoring and smells. Whenever a batch is applied, we should not consider the effect of each single refactoring rather than the effect of the batch on code smells. However, before investigating the effects of batches, we need to investigate first how to identify when a batch refactoring was performed in the source code (Research Problem 3).

**Research Problem 3.** How to identify batch refactorings is unknown.

As a matter of fact, some studies already shed some light upon the existence of batch refactorings [15, 16]. For instance, Murphy-Hill *et al.* [1] present data about batch refactorings. To determine how often programmers perform batch refactoring, they relied on programmer's action (in the IDE) to measure the temporal proximity between refactorings supported by the IDE. They say that refactorings that are performed within 60 seconds of each another form a batch. However, we do not know if temporal proximity is a realistic way identify a batch since studies report that (i) each single refactoring is often performed manually [1], and (ii) some batches are not implemented in a single programming activity, but instead in multiple change cycles [19]. In this way, heuristics should be defined to identify batches in a project (Research Problem 3). Once we are able to identify batch refactorings, we can also revisit the first two research problems in the context of batch refactorings (Research Problem 4).

**Research Problem 4.** Whether and when batches introduce, neglect, or remove smells is unknown.

Although existing studies reported the existence of batch refactorings [1, 16], there is no study that investigate the impact of batches on code smells. By tackling the research problems described in this section, we are able to better understand the relation between refactorings and smells in the context either of single or batch refactorings. The investigation of these specific problems will contribute to better understand how developers introduce or remove smells through refactoring in practice. This understanding can help us to improve assistance for smell-removal refactoring activities (General Problem).

**General Problem.** Assistance for smell-removal refactorings is limited.

Lack of sufficient knowledge of how developers apply single and batch refactorings in practice may be preventing us, researchers, from providing the required support to assist developers during refactoring operations in the context of code smells. We are not able to conceive efficient assistant techniques if we do not know: (i) how developers apply refactoring, (ii) how the code smells are affected by refactorings either positively or negatively, and (iii) how specific refactoring types affect specific types of code smells.

## 1.3 Goal and Research Questions

As discussed in the previous section, we do not have much information about the influence of refactoring on the presence of code smells. Thus, the lack of knowledge and the complexity of refactoring tasks lead us to various unaddressed research problems. Without addressing these research problems, to what extent refactoring influences on the existence of smells in the system will remain unclear. Consequently, we will not be able to provide the necessary support for developers conduct the refactoring process. In fact, these research problems are essential to provide the knowledge required to anyone understand (i) whether refactoring interferes in the density of smells, (ii) if so, how code smells are affected by either single or batch refactorings, and (iv) how to provide support for developers during the refactoring process. Given this context, the goal of this thesis is stated as follows:

**Goal.** Understand and improve batch refactorings in software systems.

To achieve this goal, we mapped the aforementioned research problems onto one or more research questions. For instance, Research Problems 1 and 2 were mapped onto two research questions, which address the problem of understanding how single refactorings are applied and how they impact the code smells existence. Although existing studies have investigated the positive impact of refactorings on code smells [17], there is still no knowledge regarding the adverse effects of refactorings on smells. Without this knowledge, it is not possible to completely understand the drawbacks related to the current refactoring practice. Therefore, to address the first research problem, our first research question is stated as follows:

**$\mathbf{RQ_1}$.** Does refactoring reduce the density of code smells?

We address this question by investigating how frequent each refactoring either removes, introduces, or neglects code smells. For this investigation, we detect refactorings that happened in the history version of real projects, then, we analyze the existence of smells in an element before and after the refactoring. First, instances of refactorings and code smells present in several software projects were detected. Then, all refactoring instances were classified according to their impact on code smells. Let $p$ the number of refactorings that were able to remove code smells; $n$ the number refactorings that introduced code smells; and $k$ representing the number of refactorings that neither removed nor introduced code smells. If $n > p$ and $n > k$, we can state that the application of refactorings are likely increasing the number of code smells of projects. Otherwise, if $p > n$ and $p > k$, the answer to our research question is **yes**, refactorings tend to remove code smells. Another possible case is when $k > p$ and $k > n$. In this scenario, refactorings would tend to neither introduce nor remove code smells.

Answer for $RQ_1$ will clarify if refactoring reduces or not the density of smells. Either way, we still have to further investigate to what extent each refactoring type interferes in the presence of smells. For instance, we may find out in the first study that, in general, refactorings reduces the density of smells. However, the reduction may not happen when we consider only a specific refactoring type, *i.e.*, it is possible that a specific type of refactoring introduces smells rather than removing them. Indeed, some types of refactoring might consistently remove (or fail to do so) or even frequently introduce specific smell types across software projects. Discovering these relations between code smells and refactoring types is the focus of our second research question:

> **$RQ_2$.** What are the patterns governing types of refactoring and code smells?

To answer this second research question, we should take into account the refactoring type as well as the smell type. Thus, we can understand and distinguish the impact of specific refactoring types on code smells. In this context, we define a pattern as a frequent relationship observed between a specific pair of refactoring type and code smell. For instance, let us assume that developers often introduce *Long Method* when applying the *Inline Method* refactoring. In this case, we say there is a pattern governing the types *Long Method* and *Inline Method*, more specifically, a pattern that indicates when the latter is applied, there is a risk of the former be introduced (details about patterns are presented in Section 2.4). Thus, by answering $RQ_2$, we are able to reveal harmful actions made by developers on code elements affected by refactorings. We detect patterns by analyzing the impact of refactoring types on smells located in the refactored elements. The knowledge about these patterns can make developers aware of the possibilities and risks of missing and introducing certain smells along either root-canal or floss refactorings.

The first two research questions consider each refactoring individually. However, in some cases, developers may apply multiple refactorings. For instance, to remove a smell such as *God Class*, a developer may need to apply a sequence of refactorings [3]. In this case, we should consider the impact of this sequence of refactorings, which we call it a batch, in the existence of smells. Hence, we will find out if developer successfully removed the *God Class* or ended up introducing other smells. The analysis of batch refactoring is indeed necessary since 40% of the times developers apply two or more refactoring operations in the same code element [1]. In this way, previous studies that consider only single refactorings can be limited in their analysis about the refactoring impact on code smells. Therefore, we need to expand our first analysis and consider the impact of batch refactorings on code smells. For this purpose, we need to find an answer for the following research question.

**RQ$_3$.** Does batch refactoring impact the density of code smells?

In order to answer RQ$_3$, we conducted a study with two-fold purposes. First, we investigate what characterizes a batch. Thus, we analyzed the data collected in previous studies to understand what constitutes a batch. Based on this analysis, we created a heuristic to automatically detect batches. Later, we detected batch refactorings in 48 software projects. After this, we observed how code smells were affected by each batch refactoring found. By analyzing the impact of batch refactorings, we might compare the results from RQ$_1$ to the ones obtained while answering RQ$_3$. In this way, we can fill the gap present in the literature about the impact of batch refactorings on code smells.

Our lack of knowledge does not restrict to not knowing the impact of batch refactorings on code smells. In fact, we do not know if there are unrevealed relationships between batches and smells. For instance, Fowler [3] states that a developer might remove a *Feature Envy* by applying a *Extract Method* followed by a *Move Method*. However, we do not know if developers do this batch, in practice, to remove *Feature Envy* instances. Furthermore, there might be specific batch refactorings that introduces code smells frequently. Perhaps, after applying a batch refactoring, developers might have to deal with newly introduced code smells frequently. Unfortunately, we do not know the patterns that associate batches and smells. Hence, in order to address this matter, the last research question of this thesis is stated as follow:

**RQ$_4$.** What are the patterns governing batches and code smells?

In order to answer RQ$_4$, we have to conduct a deeper investigation of the data used to answer RQ$_3$. The results of RQ$_3$ give us the leverage of knowing the impact of batch refactorings on code smells. After understanding the impact of batches in the density of smells, we can drill down the results to find the patterns. Consequently, we can find out what specific sequence of refactorings often introduces or removes particular code smells types. In this way, we can reveal yet unknown relations between refactorings and smells. These relations allow us to better understand the effects of refactoring on code smells when developers refactor their code in practice.

## 1.4 Scope of This Thesis

The studies conducted in the context of this thesis have, primarily, the objective of understanding how developers apply refactoring. In this way, we must define what we consider as refactoring to delimit the scope of this thesis. According to Fowler *et al.* [3], refactoring is “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.” Along with this definition, Fowler *et al.* present a catalog of code transformations defined as refactorings. As previously mentioned, examples of such code transformations, defined as refactoring types, are *Extract Method*, *Move Method*, *Extract Superclass*, and others.

Fowler *et al.* define the mechanics underlying each refactoring type, i.e., a sequence of code transformations that must be followed in order to apply the refactoring. For instance, to apply a *Move Method*, the developer must cut the method’s code from the source class, past it into the target class, and perform small changes so the code can work in its new home. Given this context, we can define what we consider as refactoring in this thesis as following.

A refactoring operation is any code transformation that matches with the mechanics defined by Fowler *et al.* independently of the developer’s intention or behavior preservation.

Although the original definition of refactoring clearly states that the code behavior must be preserved, the state-of-art shows otherwise. Murphy-Hill *et al.* [1] show that developers use the floss refactoring tactic very often. As mentioned before, floss refactoring interleaves structural improvement with other programming activities, usually changing the code behavior. For instance, developers can use floss refactoring to help to remove a functional bug. There is no way of removing a functional bug without altering the behavior. Therefore, we consider as part of the scope of this thesis refactorings that can preserve the behavior or not, since the non-preservation might happen in scenarios of bug fixing or even feature introduction.

Another particularity on what we consider as refactoring is that we disregard the developer’s intention. Since the technique we use to detect refactorings is based on matching code changes with the refactoring mechanics, we can detect and analyze refactoring operations applied involuntarily by the developers, i.e., they might not have the clear intention of refactoring the code on that change. One might say this is a major threat of this thesis since we might be answering our research questions based on code changes that were not primarily focused on refactoring. However, we believe that disregarding the developer's intention is a decision that is in synergy with the objective of this thesis.

One of the goals of this thesis is to understand how developers apply refactorings. In this vein, we are interested in understanding how refactorings affect code smells. Let us assume that we find that refactorings can degrade the code structural quality by introducing code smells. In this scenario, unintended refactorings are particularly interesting because the developers degraded the code quality through refactoring mechanics without even noticing. By studying such cases, we can learn and warn developers about such structurally-harmful refactoring operations. If we removed all refactoring operations applied without the explicit developers' intention of performing them, we would be preventing our research of discovering and reporting such interesting cases. In this way, we decided to use all refactoring operations detected regardless the developer's intention. After all, developers may be either improving or degrading the structural quality of a program if refactoring-like transformations are applied as a means to achieve other goals. In any case, in this thesis, we tried to control and distinguish the cases where refactoring-like transformations were likely applied with refactoring-driven goals in mind.

## 1.5 Main Contributions

This thesis presents studies aimed at understanding how developers perform refactoring in practice and how they affect the presence of code smells. For each one of these studies, we defined research questions, in which we found the following results:

**RQ$_1$:** In the first research question, we investigated if refactorings, in isolation, reduce the density of code smells. For that, we conducted a longitudinal study that analyzes both the beneficial and negative impact of refactoring changes on the density of smells. We analyze not only if refactoring reduces smells, but also if and to what extent specific types of refactoring are often related to the introduction of new smells. Instead of being limited to the analysis of a few major versions in a few projects, we considered 113,306 versions distributed among 23 open source projects. For each project, we classified each refactoring instance according to its interference on the existing and new smells located in the refactored elements. Hence, we classified a given refactoring instance in one of the three cases: (i) *positive* if the absolute number of smells in the elements decreases after the program transformation; (ii) *negative* if it increases; or (iii) *neutral* if it remains the same. This classification is used to analyze whether certain refactoring types tend to improve or decrease the smelly structure of a program. This analysis was also performed in samples of root-canal and floss refactorings. We found that most of refactorings are applied in code elements containing at least one code smell, which, 57% refactorings were neutral ones. Therefore, refactorings do not reduce the density of smells in the refactored elements. Even when we consider either root-canal or floss refactorings separately, refactorings do not reduce the density of code smells. Despite we found that majority of refactorings is neutral, we found that negative refactorings occur more frequently than positive ones: 33.3% against 9.7%. Also, we found that more than 95% of smells induced by refactorings were not removed afterwards in successive commits. These findings shed light on how refactorings may (in)directly or indirectly degrade the structural quality. They also suggest developers need more guidance to fully remove a code smell once they start restructuring a smelly element.

**RQ$_2$:** In the second research question, we investigated if there are yet unknown relationships between types of refactorings and code smells. We are interested in discovering scenarios where often when a refactoring type is applied, a particular code smell is introduced (or removed). For that, we used the same refactorings detected and analyzed to answer the first research question. At this point, we observed how each refactoring affected each code smell, whether they introduce or remove, and what was the code smell introduced or removed. Thus, we characterized and quantified recurring patterns governing beneficial and harmful effects of specific refactoring types. Again, harmful patterns were more frequent than beneficial ones. For instance, refactorings intended at moving methods – such as *Move Method* and *Pull Up Method* – to other classes tended to induce occurrences of *God Class* in the target classes in addition to the prevalence of smells in the source classes. The *Move Method* refactoring induced the emergence of *God Classes* in 35% of the cases, while the *Pull Up Method* tended to be related to this smell type in 28% of the cases. Moreover, 95% of such smell instances, induced by such refactorings, remained as *God Classes* in successive commits. Several other types of code refactoring were surprisingly often related to smells emerging after the program transformation. For instance, the *Extract Superclass* refactoring creates the code smell *Speculative Generality* in 68% of the cases.

**RQ$_3$:** In the third research question, we investigated if batch refactorings impact the density of code smells. In order to do such analysis, we needed first to be able to detect batch refactorings. We can think of batch refactoring as a group of single refactorings, so we had to find out how to group them in order to reveal the batches. We used adaptations of different heuristics already reported in the literature to obtain the batches. After detecting batches by considering 48 software projects, we were able to observe their impact on code smells by using a similar classification to the one presented before. In our study, we classify a given batch refactoring instance in one of the three cases: (i) *positive* if the absolute number of smells in the affected code elements decreases after the program transformation; (ii) *negative* if it increases; or (iii) *neutral* if it remains the same. Surprisingly, even when the whole batch of refactorings are considered, developers also tend to neglect or introduce more smell then remove them. More than 30% of the batches were found to be negative. Moreover, when we analyzed the commits performed after the negative batches, we also concluded that more than 95% of smells induced by batches were not removed afterwards. Only around 14% of batches removed smells.

**RQ$_4$:** In the fourth and last research question, we analyzed the patterns emerging from the relationship between batches and smells. We noticed cases of batches consistently introducing instances of *Feature Envy* in 31 different projects. This result highlights that developers consistently introduce *Feature Envy* code smells while performing batches composed of *Extract Method* refactorings. Interestingly, they also use batches of this refactoring type to remove instances of *Feature Envy*. Hence, we concluded that *Extract Methods* play a central role in the introduction and removal of *Feature Envies*. We also found that refactorings that move methods (*Move Method*, *Pull Up Method*, or *Pull Down Method*) play a central role in the removal of *God Classes*. Regarding the code smell *Complex Class*, we found that *Extract Methods* act as a complexity reducer and help developers to get rid of *Complex Classes*. Also, moving-method refactorings that use the floss-refactoring tactic contribute to the removal of such complex classes. The results obtained from answering RQ$_4$ can be used to derive common strategies used by developers during code smell removal tasks. These strategies can be used to generate heuristics for code smells removal, helping us to pave the way for improving the refactoring practice. In this sense, we derived possible heuristics for helping developers to remove *Feature Envy*, *God Class*, and *Complex Class*. Furthermore, we conducted a preliminary evaluation of such heuristics. We executed a quasi-experiment with 20 participating developers where they could evaluate if the heuristics assisted in removing code smells successfully. The results show the potential benefit of these heuristics in helping developers, leading to interesting and well-evaluated recommendations of refactorings aiming at removing code smells. By achieving a high rate of code smell removal, the heuristics might be accepted for use by developers, even though developers tend to be resistant to new techniques and tools produced by researchers [20].

## 1.6 Thesis Outline

This introductory chapter portrayed an overview of this thesis. The remainder of the thesis is structured as follows. Chapter 2 introduces an overview of basic concepts required to understand the thesis and also presents the related work. Chapter 3 presents the studies in which we collect refactorings and code smells to provide answers to $RQ_1$ and $RQ_2$. Chapter 4 presents the basic concepts regarding batch refactorings and their impact on code smells. Chapter 5 describes the methodology and presents the data needed to answer $RQ_3$ and $RQ_4$. Chapter 6 presents our preliminary study conducted to evaluate the smell-removing heuristics derived from the results of $RQ_4$. Finally, Chapter 7 concludes this thesis by summarizing the achieved research contributions, making final considerations, and pointing out directions for future research.

# 2
# Background and Related Work

The popularity of software refactoring has been increasing since the publication of the book authored by Fowler and colleagues [3]. However, the concept of software refactoring is not uniformly defined in the literature. Thus, this chapter outlines the terminology adopted throughout this thesis. It also presents three categories of studies related to ours. First, it presents studies that investigate whether smells are indicators of deep software maintenance problems. Second, presents a series of previous empirical studies conducted to investigate how and why developers perform refactoring (Section 2.5.2). Finally, it presents existing solutions proposed for supporting the identification of program elements that should be refactored (Section 2.5.3).

## 2.1 Refactoring

Software systems invariably undergo changes over the evolution that can compromise their structural quality. Thus, developers have to perform refactoring to repair the system structure. Refactoring is defined as a program transformation used for improving the structure of a program while preserving its observable behavior [3]. Code refactoring is often applied by developers during software maintenance and evolution [1].

Each refactoring can be composed of one or more elementary code transformations. From here on, each elementary transformation is named as *single refactoring operation*. Examples of single refactoring operations are *Move Method*, *Rename Method*, and *Inline Method*. Developers can apply refactoring with different purposes [21, 22], such as removing design problems, reducing maintenance effort, facilitating feature additions, improving program testability, or supporting bug fixes [15, 21, 22]. Each of these purposes can be achieved through the application of one or more refactoring operations. Regardless the specific purpose of a refactoring, a common expectation is that each refactoring operation will contribute for improving the structure of the resulting source code [23, 24, 25]. This expectation comes from the fact that refactoring, by definition, aims at improving the system structural quality. However, if refactoring operations are carried out reckless (and/or with other complementary goals), developers may (unconsciously or not) have a negative impact on the structural quality. Moreover, refactorings are often manually applied by developers [26], their negative impact may be more frequent than one could expect.

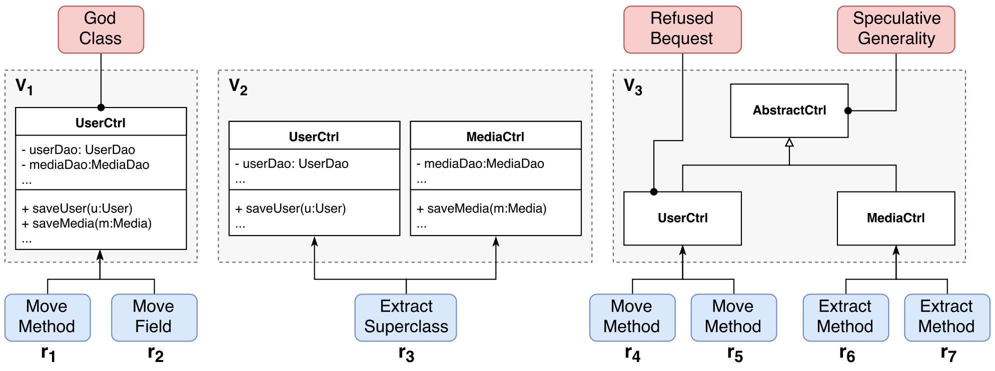


Figure 2.1: Refactoring examples. This figure presents three out of four versions of the same system ($v_1$, $v_2$, and $v_3$) changed through refactoring operations

### 2.1.1 Refactoring Characterization and Identification

Some notation must be defined before introducing the concepts of this thesis. Thus, let $S = \{s_1, \cdots, s_n\}$ be a set of software projects. Each software $s_i$ has a set of versions $V(s_i) = \{v_1, \cdots, v_m\}$. Each version $v_i$ has a set of elements $E(v_i) = \{e_1, \cdots\}$ representing all methods, classes and fields belonging to it. In Figure 2.1, the set $S = \{s_1\}$ represents the software system composed of the presented classes. This software has four versions $V(s_1) = \{v_1, v_2, v_3, v_4\}$. It is worth mentioning the $v_4$ version is the one created after performing the refactorings $r_4$, $r_5$, $r_6$, and $r_7$. Although $v_4$ is not in the figure, we consider it as part of the system versions.

Finally, each version $v_i$ has a set of elements $E(v_i)$. For instance, $E(v_2)$ is composed of *UserCtrl* and *MediaCtrl* classes, including their methods and fields. We must analyze transformations between each subsequent pair of versions to be able to identify refactorings. In this way, we assume $R$ is a refactoring identification function where $R(v_i, v_{i+1}) = \{r_1, \cdots, r_k\}$ gives us a set of refactorings. So, the function $R$ returns the set of all refactorings identified in a pair of versions.

Thus, going back to our example, $R(v_1, v_2) = \{r_1, r_2\}$. In fact, there are techniques and tools to compute the $R$ function [27, 28]. In addition to these functions, we define two more functions: $before(r_i)$ and $after(r_i)$. The first one returns the version before a refactoring, and the last one, the version after it. In the Figure 2.1, these functions provides: $before(r_1) = v_1$, and $after(r_1) = v_2$.

### 2.1.2 Refactoring History

The function $R$ can also be used to express all refactorings identified in the history of a particular system, which is composed of all refactorings identified between all subsequent pairs of versions. The refactoring history of each system is important to perform further analyses. For instance, we can use algorithms to find common patterns that emerge in the history of each software system (Chapter 3). As the function $R$ identifies all refactorings between two versions, we use its output to collect all the refactoring history of a particular system, which is composed of all refactorings identified between all subsequent pairs of versions. Since we already defined the function $R$, the refactoring history of the system $s$ can be represented by the function $H(s)$ as follow.

$$H(s) = \bigcup_{i=1}^{|V(s)|-1} R(v_i, v_{i+1}) \tag{2-1}$$

To illustrate the output of function $H(s)$, let us visit the system $s_1$ presented in Figure 2.1. This system has four versions, where three of them are represented in the figure. The fourth one is produced as the result of applying the refactorings $\{r_4, r_5, r_6, r_7\}$. Hence, $H(s_1) = R(v_1, v_2) \cup R(v_2, v_3) \cup R(v_3, v_4)$. In other words, $H(s_1)$ contains all refactorings presented in Figure 2.1, which are $\{r_1, r_2, r_3, r_4, r_5, r_6, r_7\}$.

### 2.1.3 Refactoring Type

*Refactoring type* is the kind of transformation applied in one or more code element. Thus, a refactoring type indicates if the refactoring operation is applied to attributes, methods, classes, or interfaces. Examples of refactoring types include *Move Method*, *Rename Method*, and *Extract Method*. The structural effect of refactoring operations widely varies from a refactoring type to another. For instance, code refactoring can affect: (i) multiple classes, such as in *Extract Class*, (ii) restructuring or moving methods, such as *Extract Method*, *Move Method*, and *Pull Up Method*, and (iii) extracting new code elements as in the *Extract Superclass* and *Extract Interface* [1, 8]. In order to perform our studies, we must define a list of refactoring types to be considered. Hence, we consider the most common refactoring types applied by developers [1]. Table 2.1 lists all refactoring types along with a brief description of when its is applied.

In this context, given a refactoring operation $r$, *type*$(r)$ is a function that returns the type of the refactoring $r$. In our example of Figure 2.1, *type*$(r_1)$ = *Move Method*. The identification of each refactoring type allows us to identify the effect of each specific refactoring type on the system structural quality.

## 2.1.4 Refactoring Scope

In this work, we consider as *refactoring scope* all elements directly affected by the refactoring. For instance, let us consider the *Move Method* refactoring type. In this refactoring type, a method $m$ is moved from class $A$ to $B$. Hence, the considered refactoring scope, in this case, is $\{m, A, B\}$. All callers of $m$ are indirectly affected by this refactoring, but we do not consider them as refactoring scope. Similar reasoning applies to the other refactoring types; thus, for each refactoring type, a different scope is used. In this way, given a refactoring operation $r$, $scope(r)$ is a function that returns the set of elements belonging to the scope of $r$. So, in our example, $scope(r_1) = \{saveMedia, UserCtrl, MediaCtrl\}$. Table 2.2 presents the elements considered as part of the scope for each refactoring type considered in our studies.

We should know what is the refactoring scope for each refactoring operations. The reason is that the refactoring scope (*i.e.*, the elements affected by the refactoring) is where we will look at to investigate the impact of the refactoring in the system structural quality. Therefore, we defined the scope of each refactoring type in order to minimize the effect of the code changes not related to the refactorings, i.e, by analyzing the changes in the refactoring scope, we have confidence enough to relate the changes to the refactorings.

## 2.1.5 Refactoring Tactic

As previously discussed, the goals of refactoring widely vary in practice, going from structural improvement until fixing bugs. Independently of the goal that developers want to achieve, they follow two main tactics when they refactor the source code [1]: *root-canal refactoring* and *floss refactoring*. First, the so-called *root-canal* refactoring is used for reversing the deterioration of the source code and involves a protracted process consisting of exclusive refactoring. Second, the developer employs *floss refactoring* as a means to improve structure but with the purpose of facilitating the achievement of particular objective, such as facilitating the addition of a feature, enhancing program testability, or removing a bug. Thus, in the floss refactoring, developers refactor with the intention of achieving another objective that is different from structural improvements. For example, in order to add a new feature, the developer needs to move a method to a new class; thus he needs to apply the *Move Method* refactoring before adding the new feature. Developers usually interleave floss refactoring with other types of programming activities

## 2.2 Code Smells

Previous studies [11, 29, 30, 31] have reported that refactoring apparently affects the software structure system positively. However, developers still need to know which code elements should be refactored. For this purpose, the presence of code smells has been used as a hint for refactoring [3]. A code smell is a surface indication that usually corresponds to a deeper structural problem [3, 5, 32, 33, 34]. For instance, a class with several responsibilities is known as *God Class*. This smell makes the class hard to read, modify and evolve.

To illustrate how a code smell manifests in the source code and can be a hint for refactoring, let us consider the Figure 2.1 again, which presents three versions ($v_1$, $v_2$, and $v_3$) of a particular system. In its version $v_1$, the class *UserCtrl* has, among many other members, attributes and methods implementing two loosely-coupled responsibilities: user and media management. This class accumulates two responsibilities instead of only one [35], thus, this characteristic emerges in the form of a *God Class*. To remove this smell, the developer can extract part of the class structure into another class. Therefore, the developer created the class *MediaCtrl* in version $v_2$. Also, he applied two refactorings: *Move Method*, and *Move Field*. After these transformations, the program no longer has the *God Class* and still realizes the same functionality. After these refactorings, the number of code smells was reduced.

Unfortunately, refactoring may not always successfully remove smells. Even worse, a new code smell can be introduced by refactoring. For instance, in version $v_2$, the developer thought that would be good to generalize some aspects of both classes *UserCtrl* and *MediaCtrl*. So, he applied an *Extract Superclass* refactoring in both classes, creating their superclass called *AbstractCtrl*. However, this generalization was never fully explored in the system, so the developer created a smell, called *Speculative Generality*, via refactoring. Also, the new version of *UserCtrl* uses only some of the methods and properties inherited from its parent, so the hierarchy is off-kilter. This is a code smell named *Refused Bequest*.

In this thesis, we study seventeen code smell types. These code smells were selected based on the refactoring types we use. In this way, at least one refactoring type is able to remove the considered code smells from the source code. Table 2.3 lists the code smells studied in this work.

### 2.2.1 Code Smell Identification

Several approaches have been proposed to detect code smells. These approaches include, for instance, those solely based on metrics [37, 38], based on the source code evolution information [7], based on machine learning methods [39, 40] and based on optimization algorithms such as genetic algorithms [41]. Despite the benefits or drawbacks of each approach, we have chosen to use an approach based on source code metrics [36, 37] for various reasons. Recent studies show that this approach can be used with accuracy [42, 43]. This approach has a much lower computation cost if compared to others. Also, the higher cost is not compensated by only minor accuracy improvement [44]. Moreover, we can compare our results with other studies that also used metric-based approaches [17, 45].

Metric-based approaches rely on a set of metrics and thresholds, which are combined via rules [17, 36]. These rules aim to compare the metric values with against predefined thresholds, which are combined using logical operators. In this context, we can define a function $smells(e, v)$ that returns all code smells of the element $e$ in the version $v$. Assume that exists a refactoring $r$, we can use the $smells(e, v)$ to identify the code smells in refactored elements. For this purpose, we can define the function $ScopeSmells(r, v)$ that returns all code smells existing in the refactoring scope considering the version $v$ as follow.

$$ScopeSmells(r, v) = \bigcup_{i=1}^{|scope(r)|} smells(e_i, v) \tag{2-2}$$

The $ScopeSmells$ function defined above gives us a way to obtain all code smells existing in the refactoring scope for a particular version. Once we defined a pair of versions, we can use $ScopeSmells$ to observe whether the number of smells decreased or not between both versions. Let us consider again the refactoring $r_1$ presented in Figure 2.1. We already know that $before(r_1)$ is the software version where the refactoring $r_1$ begins, and $after(r_1)$ when it ends. Therefore, $ScopeSmells(b, before(r_1)) = \{GodClass\}$ represents all existing smells in the version when the refactoring started. Similarly, $ScopeSmells(b, after(r_1)) = \emptyset$ represents the smells in the version when the refactoring ended. Given these functions, we are now able to define the next concept used in our study as follow.

### 2.2.2 Refactoring Applied in Smelly Code

We are considering each refactoring that was applied on elements that contain code smells as well as each refactoring applied on elements without code smells. In this context, a *smelly element* represents a code element (method, class, package, and the like) that contains at least one code smell. Thus, $e$ is a smelly element if and only if $smells(e, v) \neq \emptyset$. Also, assume there is a refactoring $r$. We say $r$ was applied in program elements hosting at least one code smell if any element belonging to $e$ is a smelly element, i.e., $r$ is *Applied in Smelly Code* if and only if there is a code smell of any type in the *refactoring scope*. Hence, we can define the refactoring classification scheme as follow.

## 2.3 Refactoring Classification

In this thesis, we need to conduct studies to better understand the impact of refactorings on code smells. For this purpose, we analyze how often commonly-used refactoring types affect the density of seventeen code smells. Consequently, we can classify a refactoring according to its influence on introducing a new code smell, removing an existing smell or having no effect on the number of smells. Thus, using the data returned by the functions defined before, it is possible to classify a refactoring by looking how it interferes in existing code smells. Suppose $ScopeSmells(r, before(r)) = x$, and $ScopeSmells(r, after(r)) = y$. Depending on $x$ and $y$, it is possible to classify $r$. If $x > y$, $r$ reduced the number of smells on $scope(r)$ and, because of that, $r$ is considered a *positive refactoring*. Otherwise, if $x < y$, $r$ increased the number of smells on $scope(r)$; thus, $r$ is a *negative refactoring*. When $x = y$, $r$ is classified as *neutral refactoring*.

Figure 2.1 illustrates all sorts of refactoring classifications. The refactoring $r_1$, as presented before, is classified as *positive* because its scope presented a reduction of the code smells number. In another way, the refactoring $r_3$ is negative. We classify it this way because before $r_3$ there was no code smells. However, after $r_3$ two new code smells were introduced. The $v_4$ is not present in the figure, but if we assume $r_7$ did not introduce code smells, then we can classify it as neutral.

## 2.4 Refactoring-Smell Patterns

We determine the relationship between each refactoring instance on the removal or addition of a code smell. Moreover, the refactoring classification process can also reveal to what extent and which types of refactorings often increase, rather than decrease, the number of code smells in software projects. This classification enables us to characterize recurring relationships between code smells and refactorings. For instance, if the *Move Method* refactoring introduces *God Class* frequently, it is possible to infer that there is a pattern governing these two types. We use a threshold-based rule to state a relation between refactoring and smell types as *patterns*.

Let $K = \{r_1, r_2, \cdots, r_n\}$ be the set of all detected refactorings after analyzing the set $S$. Thus, $K_{rt}$ is the subset of $K$ of refactorings of the type $rt$. The set $K^{+}_{rt,cs}$ is the $K$ subset composed of refactorings of the type $rt$ that added code smells of type $cs$ in any refactored element, while $K^{-}_{rt,cs}$ is the $K$ subset that removed code smells of type $cs$. Finally, $K^{*}_{rt,cs}$ is the $K$ subset composed of refactorings of the type $rt$ that satisfies the following conditions: (i) the refactoring was applied in classes or methods containing at least one code smell instance of the type $cs$; and (ii) the refactoring did not remove the instance of the code smell of $cs$ type.

### 2.4.1 Creational Patterns

The definition of a creational pattern between types of code smell and refactoring can be established using the above notation. A creational pattern occurs when a specific refactoring type involves code transformations that often introduces a specific code smell. We define this concept as a threshold-based rule. If $|K^{+}_{rt,cs}|/|K_{rt}| \geq \gamma$, it is possible to affirm that there is a creational pattern between $rt$ and $cs$. This kind of pattern captures scenarios where developers apply a refactoring and, somehow, end up creating at least one new code smell. Thus, creational patterns represent cases of *stinky refactorings*. We named these refactorings as stinky ones since they lead to the introduction of code smells, *i.e.*, they have a negative impact on the presence of smells.

### 2.4.2 Removal and Non-Removal Patterns

The definition of *removal pattern* also lies in a threshold-based rule. If $|K^{-}_{rt,cs}|/|K_{rt}| \geq \gamma$, we can affirm that there is a removal pattern between $rt$ and $cs$. It means that developers consistently removes instances of $cs$ when performing $rt$ refactorings. We are also interested in studying what types of code smells are commonly present in classes and methods and, somehow, end up remaining in the source code after refactoring. This third type of pattern is called *non-removal pattern* and it is defined by another threshold-based rule: $|K^*_{rt,cs}|/|K_{rt}| \geq \gamma$.

## 2.5 Related Work

This thesis resides in a territory between refactoring and code smells. Several studies aimed at understanding how and why developers perform refactoring. Since code smells motivate refactoring operations [3], many studies were conducted to understand how code smells can reflect deeper problems in software projects. Also, many studies aimed at identifying pieces of code that require refactoring as a means to support developers during software maintenance tasks. In this section, we present several studies distributed on three main topics. First, in Section 2.5.1, we present studies focused on understanding how problematic code smells can be. Second, in Section 2.5.2, we present studies that contribute to the understanding of the refactoring practice. Last, in Section 2.5.3, we discuss the state-of-art regarding the identification of refactoring opportunities.

### 2.5.1 Code Smells as Symptoms of Deeper Problems

Fowler and colleagues [3] were the first to propose the notion of code smells. According to them, code smells are indicators of deeper problems in the software systems. Since the publication of their work, many studies were conducted to investigate the real relevance and impact of code smells in different levels of structural quality of software systems.

Macia *et al.* conducted many studies that show how code smells are indicators of architectural design degradation [5, 32, 46, 47]. Research has shown that the longevity of evolving software systems largely depends on their resilience to architectural design degradation [47]. Before these studies, there was still limited knowledge about the circumstances under which code smells represent architectural problems. Therefore, without this knowledge, developers would have a hard time to implement architecturally-relevant strategies for code refactoring. Macia *et al.* executed a series of empirical studies about the influence of code smells on architecture degradation symptoms. One of these studies [5] aimed at understanding the relationship between code smells and architecture problems in 6 software systems, which were intended to adhere different architectural decomposition. The authors reported that 78% of all architecture problems in the programs were related to code smells. They also found that the refactoring strategies, even when frequently applied in those systems, did not significantly contribute to removing architecturally-relevant code smells.

In a similar vein, many studies were conducted to investigate the correlation between code smells and design problems [9, 34, 48]. Similarly to architectural problems, design problems are often villains responsible for the discontinuation or redesign of software systems [9]. As design documentation is often informal or nonexistent, design problems need to be located in the source code. The main difficulty to identify a design problem in the implementation stems from the fact that such issue is often scattered through several program elements [9]. While Macia *et al.* [5] used each code smell alone to identify architectural degradation, Oizumi *et al.* hypothesize that code smells tend to "flock together" to realize design problems. The authors analyze to what extent groups of inter-related code smells, named agglomerations, suffice to locate design problems. After examining more than 2,200 agglomerations scattered through seven software systems, they conclude that specific forms of agglomerations are consistent indicators of both congenital and evolutionary design problems, with accuracy often higher than 80%.

Many studies focused on investigating the impact of code smells on software maintainability. Khomh *et al.* [40] have investigated the relation between 29 code smells and changes occurring in classes from two software projects. Their results showed that classes with code smells are more likely to change than classes without a smell. Palomba *et al.* [25] capture previous findings on bug-proneness to build a specialized bug prediction model for smelly classes. Specifically, they evaluate the contribution of a measure of the severity of code smells by adding it to existing bug prediction models and comparing the results of the new model against the baseline model. Their results indicate that the accuracy of a bug prediction model increases by adding the code smell intensity as a predictor.

Yamashita and Moonen [10, 49, 50] conducted many studies for exploring the impact of inter-smell relations on software maintainability. While previous studies mainly focused on the effects of individual code smells on maintainability, they conjecture that not only the individual code smells but also the interactions between code smells affect maintenance. Yamashita and Moonen [49] empirically investigate the interactions amongst 12 code smells and analyze how those interactions relate to maintenance problems. They observed that code smells co-located in the same artifact interacted with each other, and affected maintainability.

Tufano *et al.* [51] executed a large empirical study using change history of 200 open source projects from different software ecosystems. This study had two objectives: (i) to investigate when developers introduce code smells; (ii) and what are the circumstances and reasons behind their introduction. The authors developed a strategy in order to identify commits, which introduced code smells. Using this strategy, they mined over 0.5M commits and did manual analysis of 9,164 commits identified as smell introducing. They found smells are generally introduced during improvement of existing features or during the implementation of new ones.

As discussed, many studies were successful in showing how code smells are symptoms of deeper problems in software systems, such as architectural, design, and maintainability issues. Also, we mentioned a study that affirms the abundance of code smells often lead to the increase of change-proneness of classes. Therefore, the developers should fight against the presence of code smells. Also, developers must be constantly aware of the possibility of introducing new ones, since they might lead to more severe problems. In this context, our first research question investigates whether refactorings are capable of removing or even introducing smells. Given the problems code smells suggest, developers must avoid to add them in their source code, mainly when refactoring – which is an operation intended to improve the code structure.

### 2.5.2 Studies of Software Refactoring

Some studies were conducted in order to understand how and why developers apply refactoring [16, 28]. Murphy-Hill *et al.* [1] draw conclusions using a large data set about the refactoring practice. First, they report refactorings are performed frequently. Second, almost 90% of refactorings are performed manually, without the help of tools. Finally, they say about 40% of refactorings performed using a tool occur in batches. Given the high frequency of manually-performed refactorings, Ge *et al.* [52] designed a tool called BeneFactor. BeneFactor detects a manual refactoring, reminds the developer that automatic refactoring is available, and can complete her refactoring automatically. BeneFactor is designed to help solve the refactoring tool underuse problem.

Regarding what motivates developers to perform refactoring, Silva *et al.* [15] investigated the reasons that drive developers to refactor their code. They identified refactorings on 748 Java projects in the GitHub repository. Then, they asked developers why they performed the identified refactorings. Their results indicate that fixing a bug or changing the requirements, such as feature additions, mainly drives refactorings. Their results show that the refactored code may contain code smells, although developers did not mention it explicitly as the intention to refactor. Wang [53] also contributes to the understanding of the motivations behind developers. Through interviews with 10 professional software developers, Wang identified the major factors that motivate their refactoring activities. Some of the reported findings are consistent with the ones reported by Silva *et al.* [15].

Bavota *et al.* [17] mined the evolution history of 3 Java open source projects to investigated if refactorings occur on code components that certain indicators suggest a need for refactoring. Their considered indicators include structural quality metrics and the presence of code smells. They also measure the e effectiveness of refactorings regarding their ability to remove code smells. According to their results, quality metrics do not show a clear relationship with refactoring and 42% of the refactorings are applied on code elements with code smell, in which only 7% of them remove smells. Stroggylos and Spinellis [2] analyzed source code of popular open source software systems to detect refactorings and examine how the software metrics are affected by this process. They evaluate whether refactoring is effectively used as a means to improve software quality. Although it would be expected that the increase in quality achieved via refactoring is reflected in the various metrics, measurements on real-life systems indicate the opposite.

Some studies were conducted to understand the impact of refactorings on software defects. Fujiwara *et al.* [13] and Ratzinger *et al.* [11] investigated the influence of software changes, such as refactoring, on bug fixes required in later versions. They studied whether refactoring reduces the probability of software defects and whether refactoring is more important than bug fixing for software quality. The authors found that an increase in refactoring has a significant positive interference on software quality. Results showed that number of software defects in the target period decreases if more refactorings are applied. They also observed a defect decrease if these refactorings increase compared to bug fixes. Differently, Soares *et al.* [54, 55] and Bavota *et al.* [17] show the refactoring tools can introduce bugs by changing the behavior of the refactored code. Results indicate that, while some kinds of refactorings are unlikely to be harmful, others, such as refactorings involving hierarchies (e.g., pull up method), tend to induce faults very frequently [17].

The studies presented in this section show that refactorings are used very often during software development projects. However, the developers usually apply refactorings with no tooling support. This practice might increase the risks involved with refactoring operations. Although some studies show refactorings help to prevent defects, some studies show that refactorings can introduce bugs. Hence, refactorings can also have an adverse impact either in the code expected behavior or even in the structure of the code. Since refactorings can also pose a threat to the code structure, code smells can also be introduced inadvertently. Again, this fact is closely related to our first research question. In this way, our study can help to increase the awareness of the researchers and developers about possible risks of introducing code smells while refactoring.

### 2.5.3 Identification of Refactoring Opportunities

The identification of program structures that should be refactored is particularly difficult for large projects, which lead studies to investigate how to support developers in the identification of refactoring opportunities. Usually, these studies focus on specific types of refactorings to identify the opportunities [56]. Also, they use different techniques to determine such refactoring opportunities, such as quality metrics, machine learning models, and data flow analysis. In this section, we compile some studies aiming at identifying refactoring opportunities by searching for poor structures that urge for refactoring.

Tsantalis *et al.* [56] propose a methodology for the identification of *Move Method* refactoring opportunities that constitute a way for solving many common *Feature Envy* code smells. The authors present an algorithm that employs the notion of distance between system entities (attributes/methods) and classes. This algorithm extracts a list of behavior-preserving refactorings based on the examination of a set of preconditions. The authors also published similar techniques [57, 58, 59] that aim at identifying refactoring opportunities of *Extract Method* and *Polymorphism Introduction.*

Al Dallal [60] explores several quality metrics considered individually and in combination to predict the classes in need of *Extract Subclass* refactoring. For a given class, the author empirically investigates, using univariate logistic regression analysis, the abilities of 25 existing size, cohesion, and coupling metrics to predict whether the class is in need of restructuring by extracting a subclass from it. The results indicate that there was a strong statistical relation between some of the quality metrics and the decision of whether *Extract Subclass* refactoring was required. This work uses only metrics values in order to identify refactoring opportunities and, in this case, the author was successful in the identification of *Extract Subclass* opportunities. The author did not use his model to identify opportunities for other types of refactoring.

Bavota *et al.* [61, 62] present a technique to identify *Extract Class* refactoring opportunities. They propose a refactoring method based on graph theory that exploits structural and semantic relationships between methods. The empirical evaluation of the proposed approach [61] highlighted the benefits provided by the combination of semantic and structural measures and the potential usefulness of the proposed method as a feature for software development environments. Bavota *et al.* [62] also propose a different technique to recommend *Extract Class* refactoring opportunities based on game theory. Pappalardo and Tramontana [63] propose a technique based on the measuring strength of method interactions to identify *Extract Class* refactoring opportunities.

In a different vein, Bois *et al.* [64] analyze how refactorings manipulate coupling and cohesion characteristics, and how to identify refactoring opportunities that improve these characteristics. Similarly, Meananeatra [65] focuses on identifying refactorings for improving software maintainability. Panita Dietrich *et al.* [66] present a technique to detect starting points for refactoring of large and complex systems based on the analysis and manipulation of the type dependency graph extracted from programs. Other authors [67, 68] also use the dependency graph to propose find different refactoring opportunities, such as the *Template Method* refactoring type.

Researchers use different techniques and propose several methods to find refactoring opportunities. However, developers still do not use refactoring tools to support daily maintenance tasks. Also, those techniques do not help developers to prevent the creation of problems through refactorings, such as quality metrics degradation [45], and defect introduction [17]. In this way, there is a long way to run in what concerns helping developers to identify refactoring opportunities. Our second research question is one step towards the better understanding of the refactoring practice. By understanding common patterns applied by developers, we might be able to help developers to identify refactoring opportunities and prevent common mistakes. For instance, the introduction of code smells.

## 2.6 Summary

The concept of software refactoring is not uniformly defined in the literature. In this way, this chapter presents a conceptual framework that will be used throughout this thesis. In this way, we can formalize and motivate the next chapters without ambiguity. Also in this chapter, we presented studies that relate to our own. As a consequence, we were able to identify gaps in the literature that can be filled by this thesis.

As discussed before, Little is known about the impact of refactoring over code smells. Bavota *et al.* [17] only investigated, in a few projects, the capability of refactorings for removing code smells. However, nothing was reported about the cases where refactorings introduce code smells. Consequently, little is known about refactoring types that commonly introduce specific types of code smells, revealing ordinary harmful patterns applied by developers. In this way, there is opportunity to research in these two topics. The next chapter presents our efforts to fill these gaps.

Table 2.1: Refactoring types

| **Type** | **Description** |
|---|---|
| Extract Interface | This transformation is applied when several clients use the same subset of a class's interface, or two classes have part of their interfaces in common. Then, this transformation aims at extracting the subset into an interface. |
| Extract Superclass | This transformation is applied when two classes have similar features. The transfomation is used to create a superclass, which it is moved the classes common features to the superclass |
| Extract Method | This transformation is applied when a code fragment can be grouped together. Then, this transformation aims at turning the fragment into a method, then a name is created to explain the purpose of the method. |
| Inline Method | This transformation is applied when a method body is more obvious than the method itself. Then, this transformation replaces calls to the method with the method's content and deletes the method itself |
| Move Method | This transformation is applied when a method is used more in another class than in its own class. Thus, the transformation creates a new method in the class that uses the method the most, then it moves code from the old method to there. Finally, it turns the code of the original method into a reference to the new method in the other class or else remove it entirely |
| Pull Up Method | This transformation is applied when methods have identical results on subclasses. Then, it movse them to the superclass. |
| Push Down Method | This transformation is applied when behavior on a superclass is relevant only for some of its subclasses. Then, it moves it to those subclasses. |
| Rename Method | This transformation is applied when the name of a method does not reveal its purpose. Then, it changes the name of the method. |
| Move Field | This transformation is applied when a field is, or will be, used by another class more than the class on which it is defined. Then, it creates a new field in the target class, and it changes all its users. |
| Pull Up Field | This transformation is applied when two subclasses have the same field. Then, it moves the field to the superclass. |
| Push Down Field | This transformation is applied when a field is used only by some subclasses. Then, it moves the field to those subclasses. |

Table 2.2: Refactoring scope

| **Type** | **Refactoring Scope** |
|---|---|
| Extract Interface | Classes implementing the new interface. The new interface is not part of the scope because it has no source code other than the method signatures. |
| Extract Superclass | Classes extending the new class; and new class created. |
| Extract Method | The method created; the method from where the new method was extracted; and class containing both methods. |
| Inline Method | The method which received the new code; and class containing the method. |
| Move Method | The two classes affected by the change: the class which the method used to reside and the class which received the method. |
| Pull Up Method | The two classes affected by the change: the class which the method used to reside and the class which received the method. |
| Push Down Method | The two classes affected by the change: the class which the method used to reside and the class which received the method. |
| Rename Method | The renamed method and the class that contains it. |
| Move Field | The two classes affected by the change: the class which the field used to reside and the class which received the field. |
| Pull Up Field | The two classes affected by the change: the class which the field used to reside and the class which received the field. |
| Push Down Field | The two classes affected by the change: the class which the field used to reside and the class which received the field. |

Table 2.3: Smell types

| **Smell Type** | **Description** |
|---|---|
| Brain Class | This design disharmony is about complex classes that tend to accumulate an excessive amount of intelligence, usually in form of several methods affected by *Brain Method* [36]. |
| Brain Method | Long and complex method that centralizes the functionality of a class. |
| Class Data Should be Private | Class that exposes its fields, violating the principle of data hiding. |
| Complex Class | A class having at least one method having a high cyclomatic complexity. |
| Data Class | A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes do not contain any additional functionality and cannot independently operate on the data that they own [36]. |
| Dispersed Coupling | A method that accesses many code elements, and the accessed code elements are dispersed among many classes. |
| Feature Envy | A method that is more interested in a class other than the one it actually is in. |
| God Class | When a class accumulates several responsibilities. |
| Intensive Coupling | A method that has tight coupling with other methods, and these coupled methods are defined in the context of few classes. |
| Lazy Class | A class having few lines of code, few methods, and with low complexity. |
| Long Method | A method that is unduly long in terms of lines of code. |
| Long Parameter List | A method having a long list of parameters, some of which avoidable. |
| Message Chain | A long chain of method invocations is performed to implement a method functionality. |
| Refused Bequest | A class redefining most of the inherited methods, thus signaling a wrong hierarchy |
| Shotgun Surgery | Resembles *Divergent Change*, but is actually the opposite smell. *Divergent Change* is when many changes are made to a single class. *Shotgun Surgery* refers to when a single change is made to multiple classes simultaneously. |
| Spaghetti Code | A class implementing complex methods interacting between them, with no parameters, using only attributes to exchange data. |
| Speculative Generality | An abstract class having very few children classes using its methods. |

# 3
# Investigating the Impact of Refactorings on Smells

Refactoring, a common practice employed by developers along software maintenance [1, 2], is a program transformation used for improving the structure of a program while preserving its observable behavior [3]. In order to achieve the structural improvement, developers need know when and where to apply refactoring. In this context, code smells, which are indicators of software structural problems (Section 2.2), appear as an indicator of refactoring opportunities. In other words, if a code element contains smells, then it is a candidate for refactoring. Hence, a reasonable assumption is to expect that when developers refactor smelly elements (*i.e.*, elements with code smells), they reduce the density of code smells. Unfortunately, to what extent such assumption holds in practice is unknown. The reason for that is our little knowledge about the impact of refactoring on the existence of smells.

Unfortunately, the impact of refactoring on smells is rarely investigated in depth. To the best of our knowledge, there is no study that thoroughly characterizes both positive and negative effects of refactoring on code smells. Bavota *et al.* [17] performed a study aiming to investigate if refactoring tends to remove code smells in the context of some major versions of only three software projects. However, they could not reveal to what extent and which types of refactorings often increase, rather than decrease, code smells in a program. Moreover, they did not make a distinction between root-canal and floss refactoring in their analyses (Section 2.1.5). Finally, given the nature and size of their sample, they could not characterize recurring relationships between refactoring and smell types.

Thus, we conduct a longitudinal study that analyzes both the beneficial and negative impact of refactoring on the density of smells. We analyze not only if refactoring reduces smells, but also if and to what extent specific types of refactoring are often related to the introduction of new smells. Instead of being limited to the analysis of a few major versions in a few projects, we consider 113,306 versions (commits) distributed among 23 open source projects to conduct the study. Since we aimed at conducting a large-scale study with several projects, we had to conduct a preliminary study with only smaller projects to set up a suitable methodology. Consequently, this preliminary study was extremely important for testing and improving our methodology and tools we develop to collect and process the outcomes. Hereupon, we classify each refactoring instance according to its interference on the existing and new smells located in the refactored elements. In our study, we classify a given refactoring instance in one of the three cases: (i) *positive* if the absolute number of smells in the elements decreases after the program transformation; (ii) *negative* if it increases; or (iii) *neutral* if it remains the same (Section 2.3). This classification is used to analyze whether certain refactoring types tend to improve or decrease the smelly structure of a program. This analysis was also performed in samples of root-canal and floss refactorings.

We identified and analyzed 51,461 refactorings classified in 10 commonly used [1] refactoring types. Thirteen code smell types are used to classify the collected refactorings (Section 3.1.2.2). These code smell types were selected because they are conceptually associated with the definition of the refactoring types [3], *i.e.*, the definition of each refactoring type is explicitly associated with one or more code smells addressed in our study. Surprisingly, our study revealed that either neutral or negative effects of software refactoring are much more frequent than positive effects.

This chapter is organized as follows. Section 3.1.1 provides the research questions that guide this study. Section 3.1.2 presents the study planning. Section 3.2 presents the results of the preliminary study. Sections 3.3 and 3.4 present the results of the main study. Section 3.6 describes the threats to validity. Section 3.7 relates our study with previous work. Finally, this chapter is summarized in Section 3.8.

## 3.1 Settings of the Study

This section describes the settings of the study conducted to investigate the impact of refactorings on code smells . In particular, Section 3.1.1 presents the goal of the study and the questions addressed, and Section 3.1.2 details the design of the study.

### 3.1.1 Goal and Research Questions

Although existing studies have investigated the positive impact of refactorings on code smells [17], there is still no knowledge regarding the adverse effects of refactorings on code smells. Without this knowledge, it is not possible to completely understand the drawbacks related to the current refactoring practice. In this context, the goal of this study is stated as follows:

**Goal:** Analyze refactoring operations for understanding whether and how code smells are introduced through refactoring.

We have to achieve this goal in order to better understand the relation between code smells and refactoring. As previously mentioned, developers can use the presence of code smells as a hint to apply refactoring. Thus, this relation exists since the former is used to indicate the latter. In fact, code smells are symptoms of structural problems [5, 32, 46, 47, 48]. Since the goal of refactoring is to improve the system structural quality, one may expect that when developers refactor smelly elements, they remove smells; thus, improving the structural quality. Indeed, software refactoring might interfere in the presence of code smells. As illustrated in Section 2.1, the number of code smells located in refactored elements should be ideally reduced. However, we do not know if in practice that happens. Refactoring is a complex task, which requires the developer to be aware of its impact. Thus, if refactorings are carried out reckless, developers may introduce smells instead of removing them. Therefore, the following question was refined from the previous goal:

**$\mathbf{RQ_1}$.** Does refactoring reduce the density of code smells?

We address this question by relying on the classification of each refactoring detected in real projects. This procedure enables us to compute how frequent each refactoring classification occurs across the projects. First, all instances of refactorings and code smells present in a set $S$ of software were detected. Then, all refactoring instances were classified according to the scheme presented in Section 2.3. Also, we divided the refactorings into root-canal and floss refactoring (Section 2.5.2). Thus, we were able to compute if refactoring reduces or not the density of smells. For this computation, consider $p$ to be the number of refactorings classified as positive; $n$ the number of negative refactorings; and $k$ representing the number of neutral refactorings. If $n > p$ and $n > k$, we can state that the application of refactorings are likely increasing the number of code smells of projects. Otherwise, if $p > n$ and $p > k$, the answer to our research question is **yes**, refactorings tend to remove code smells. Another possible case is when $k > p$ and $k > n$. In this scenario, refactorings would tend to neither introduce nor remove code smells.

Answering for $\mathrm{RQ_1}$ will allow us to understand, in general, the impact of refactoring on the existence of smells. However, to understand and distinguish the impact of specific refactoring types on code smells is also important. Some types of refactoring might consistently remove (or fail to do so) or even frequently introduce specific smell types across software projects. Knowing these removal and creational patterns can help researchers to propose techniques that can help developers to avoid creational patterns while removing code smells. Regarding this matter, Section 2.4 defined three categories of possible patterns between types of refactoring and smells: creational, removal and non-removal patterns. Discovering these patterns is the focus of our second research question:

> **RQ$_2$.** What are the patterns governing types of refactoring and code smells?

By answering RQ$_2$, we are able to reveal harmful actions made by developers on refactored elements. We detect removal, non-removal and creational patterns by analyzing the impact of refactoring types on smells located in the refactored elements. The knowledge about non-removal and creational patterns make developers informed about the possibilities and risks of missing and introducing certain smells along either root-canal or floss refactorings.

### 3.1.2
### Study Phases

To achieve the goal, and answer the questions defined in Section 3.1.1, this study analyzed refactorings in open source projects. In this section, we present all the phases of this study. As mentioned at the beginning this very same chapter, we first conducted a preliminary investigation to test and improve our experimental design. This section presents the resulting study design after all improvements performed as the outcome of the preliminary study. We additionally show the results and lessons learned from the initial preliminary study in Section 3.2.

#### 3.1.2.1
#### Phase 1: Selection of Software Projects

The first step of this study is to choose a set $S$ of software projects to compose the study sample. First, we established GitHub, the world's largest open source community, as the source of software projects. We focused our analysis on open source projects so that our study could be easily replicated and extended. This study uses 23 GitHub projects that met the following quality criteria:

– Projects with different levels of popularity. To meet this criterion, we used the number of Github stars in each project to measure its popularity level. GitHub star is a metric to keep track of how popular an open source project is among users;

– An active issue tracking system, i.e., users actively use the GitHub issue management system for bug reporting and improvement suggestions

– Projects with at least 90% of the code repository effectively written in Java.

We selected software projects based on these criteria because they allow us to select projects with different structure, size and popularity. Additionally, we selected Java projects due the popularity of the Java programming language[1]. This table presents the project (i) domain, (ii) name, (iii) lines of code, (iv) number of classes, (v) number of commits and (vi) starts for each project.

Table 3.1: Projects used

| **Domain** | **Project** | **LOC** | **Number of classes** | **Commits** | **Stars** |
|---|---|---|---|---|---|
| Android | Facebook Fresco | 50,779 | 860 | 744 | 14,679 |
| | OkHttp | 49,739 | 642 | 2,645 | 27,421 |
| | Google I/O Sched App | 40,015 | 754 | 129 | 15,686 |
| | PhilJay MPAndroidChart | 23,060 | 268 | 1,737 | 23,036 |
| | Dagger | 8,889 | 441 | 696 | 11,097 |
| | Android Bootstrap | 4,180 | 123 | 230 | 4,298 |
| | LeakCanary | 3,738 | 127 | 265 | 19,847 |
| | Orhanobut Logger | 887 | 11 | 68 | 9,423 |
| Application | Google J2ObjC | 385,012 | 4,866 | 2,823 | 5,172 |
| | ArgoUML | 177,467 | 2,597 | 17,654 | 5 |
| | Apache Ant | 137,314 | 1,784 | 13,331 | 205 |
| Database | Presto DB | 350,976 | 4,146 | 8,056 | 7,740 |
| | Realm Java | 50,521 | 1,018 | 5,916 | 9,682 |
| Framework | Spring Framework | 555,727 | 12,715 | 12,974 | 22,052 |
| | Apache Dubbo | 104,267 | 1,690 | 1,836 | 19,934 |
| | JUnit4 | 26,898 | 1,251 | 2,113 | 6,935 |
| Library | Elasticsearch | 578,561 | 8,845 | 23,597 | 32,200 |
| | Spring Boot | 178,752 | 5,178 | 8,529 | 26,294 |
| | JBoss Xerces | 140,908 | 1,136 | 5,456 | 4 |
| | Facebook SDK for Android | 42,801 | 836 | 601 | 4,534 |
| | Netflix Hystrix | 42,399 | 1,569 | 1,847 | 14,172 |
| | Retrofit | 12,723 | 554 | 1,349 | 26,557 |
| Web Application | Netflix SimianArmy | 16,577 | 244 | 710 | 6,618 |
| | **Total** | **2,982,190** | **51,655** | **113,306** | **307,591** |

### 3.1.2.2
### Phase 2: Smell and Refactoring Detection

This phase is in charge of detecting refactorings in all subsequent pairs of versions $v_i$ and $v_{i+1}$. It also encompasses the detection of all smells in each version $v_i \in V(s)$. These activities are described in the following.

**Refactoring Detection.** We need a tool to detect refactorings between a pair of subsequent versions (Section 2.1.1). For this purpose, we choose Refactoring Miner [28, 69] to support the detection of refactoring instances. This

[1]`https://www.tiobe.com/tiobe-index/`

tool implements a lightweight version of UMLDiff [70] algorithm for differencing object-oriented models. The precision of 96.4% reported by Tsantalis *et al.* [28] led to a very low rate of false positives, as confirmed in our validation phase (Section 3.1.2.4). This tool supports the detection of all refactoring types described in Section 2.1.3, which are amongst the ones reported by Murphy-Hill *et al.* [1] as the most common refactoring types. All refactoring types detected by Refactoring Miner were considered in this study, except the *Rename Method* refactoring. We discarded this refactoring type as it was not directly related to one of the code smells addressed in our study. Refactoring Miner gives us as output a list of refactorings $R(v_i, v_{i+1}) = \{r_1, \cdots, r_k\}$ as defined before, where $k$ is the total number of refactorings identified.

**Code Smell Detection.** Code smells are often detected with metric-based strategies [46]. Each strategy is defined based on a set of metrics and thresholds. Thus, the application of metric-based strategies requires the collection of metrics for all source files in a project. After the collection of metrics, we apply a set of previously defined rules [5, 36] to detect code smells. This procedure is the implementation of *smells* function defined in Section 2.2.1. The specific metrics and thresholds for code smell detection were defined in [5, 47]. These rules were used because: (i) they represent refinements of well-known rules proposed by Lanza *et al.* [36], which are well documented and used in previous studies (*e.g.*, [50, 71]); and (ii) they have, on average, precision of 0.72 and recall of 0.81 [72].

These rules detect five code smells: *God Class*, *Long Method*, *Feature Envy*, *Shotgun Surgery* and *Divergent Change*. Table 3.2 shows examples of rules used to identify code smells. The rules use the following metrics: (i) Lines of Code — LOC; (ii) Coupling Between Objects — CBO; (iii) Number of Methods — NOM; (iv) Cyclomatic Complexity — CC; (v) Lack of Cohesion of Methods — LCOM; (vi) Fan-out — FO; and (vii) Fan-in — FI. We also considered eight additional smell types: *Complex Class*, *Lazy Class*, *Long Parameter List*, *Message Chain*, *Refused Bequest*, *Spaghetti Code*, *Speculative Generality*, and *Class Data should be Private*.

Table 3.2: Rules for code smell detection

| **Code Smell** | **Detection Rule** |
|---|---|
| God Class | [(LOC $>\alpha$) and (CBO $>\beta$)] or [(NOM $>\delta$) and (CBO $>\beta$)] |
| Long Method | (LOC $>\epsilon$) and (CC $>\eta$) |
| Shotgun Surgery | (CC $>\theta$) and (FO $>\omega$) |
| Divergent Change | (FI $>\iota$) and (LCOM $<\upsilon$) and (CC $>\varsigma$) |
| Feature Envy | (FO $>\zeta$) and (LCOM $<\varrho$) and (CC $>\varpi$) |

We selected these smell types because they are very common and tend to be related to design degradation symptoms [5]. Another reason to select these smell types was their direct relation with the most frequent refactoring types [3]. Indeed, Murphy-Hill *et al.* [1] reported which are the refactorings often performed by developers, and we analyzed which code smell types these refactorings may intend to remove [3]. Refactoring Miner is capable of detecting the refactoring types that remove these smells. Unfortunately, we could not find any publicly available tool to detect all 13 smell types. Thus, we implemented a tool to detect these mentioned types of code smells [73].

Table 3.3: Code smell detection rules proposed by Bavota *et al.*

| **Code Smell** | **Rule** |
|---|---|
| Class data should be private | A class having at least one public field. |
| Complex class | A class having at least one method for which McCabe cyclomatic complexity is higher than 10. |
| Feature envy | All methods having more calls with another class than the one they are implemented. |
| God class | All classes having (i) cohesion lower than the average of the system AND (ii) LOCs >500. |
| Lazy class | All classes having LOCs lower than the first quartile of the distribution of LOCs for all system's classes. |
| Long method | All methods having LOCs higher than the average of the system. |
| Long parameter list | All methods having a number of parameters higher than the average of the system. |
| Message chain | All chains of methods' calls longer than three. |
| Refused bequest | All classes overriding more than half of the methods inherited by a superclass. |
| Spaghetti code | A class implementing at least two long methods (see previous rule) interacting between them through method calls or shared fields. |
| Speculative generality | A class declared as abstract having less than three children classes using its methods. |

In addition to the rules presented in Table 3.2, we also implemented all rules proposed in a recent study [17] to detect the smells, as presented in Table 3.3. These rules detect eight additional types of code smells. Rules for detecting smells play a central role in our study. Thus, we must guarantee that our results are not biased by a single set of detection rules. Different thresholds can lead to different results [44, 74]. Therefore, choices of thresholds can pose a threat to this study. Thus, two sets of thresholds were used to mitigate this menace. The first set, known as *tight* set, represents the thresholds previously validated in the study by Macia *et al.* [5]. We named this strategy as *tight* because it relies on the use of high threshold values aiming to detect only critical code smells across the projects. The second strategy, named as *relaxed*, uses relaxed thresholds designed to detect as many smells as possible. In addition to the two previously mentioned (tight and relaxed) set of thresholds, we also used the detection rules proposed by Bavota *et al.* [17].

### 3.1.2.3
### Phase 3: Refactoring Classification

The objective of the third phase is to classify all refactorings detected in the prior phase. We classified each detected refactoring by observing its interference in the number of code smells. After this classification, it is possible to quantify how frequent refactorings are labeled according to each possible category in our software set $S$. As mentioned in Section 3.1.2.1, all projects are Git repositories stored on GitHub servers. The data collection process starts by cloning a Git repository. This study considers as a version every commit in the repository. We skipped merge commits during the analysis since this kind of commit could lead us to compute twice the same refactoring [28]. The algorithm always compares subsequent versions of the projects. Let us suppose a project that has only three commits: 1, 2, and 3. In this project, the R function would be computed for the following pairs: $R(1,2)$ and $R(2,3)$. The set $V(s)$ of a Git repository $s$ is the list of all non-merge commits in the master branch ordered chronologically.

As described in Section 2.1.5, there are two refactoring tactics: (i) root-canal refactoring; and (ii) floss refactoring. In this way, this study comprises a manual inspection of a randomly selected sample of refactorings. In this manual inspection, we evaluate if a refactoring is root-canal or floss. We analyzed manually whether the changes performed during the refactoring do not modify the behavior. We classify a transformation as floss when we identify behavioral changes, such as an addition of methods or changes in a method body not related to refactoring transformations. When no behavioral changes are detected, we classify the refactoring as root-canal. This manual inspection will enable us in revealing the percentage of positive, negative and neutral effects in the context of both root-canal and floss refactorings. This inspection was performed by three researchers. Two of them are very experienced refactoring researchers. The most experienced one solved the conflicts. We found that developers apply root-canal refactoring in 31.5% of the cases. The confidence level for this number is 95% with a confidence interval of 5%.

#### 3.1.2.4 Phase 4: Manual Validation

The last phase is responsible for the data validation. As the first three phases rely on a tool to detect refactorings, there is a threat to validity related to false positives and negatives yielded by it. To mitigate this threat, this fourth phase is required. In this phase, a manual procedure was executed in a smaller dataset. A manual validation of each refactoring type and tactic was made in this phase to ensure the reliability of our data. In this vein, we conduct different data validation activities.

The first validation was regarding the refactoring type. So, we randomly sampled refactorings from each type to support the analysis. We decided to sample by the 10 refactoring types since the precision of the Refactoring Miner could vary due to the rules implemented in the tool to detect each refactoring type. Consequently, the tool could have a high precision for an specific type while achieving a low precision for another one; thus, the need for such validation. To ensure an acceptable confidence level in the results, we calculated the sample size of each refactoring type based on a confidence level of 95% and a confidence interval of 5 points. We used such confidence to all sampling activities performed in this study. We recruited ten undergraduate students from another research group to also analyze the samples. The samples were divided into ten disjointed sets, and each student validated a different one. In general, it was observed a high accuracy for each refactoring type, with a mean of 88.36 %. We highlight that we relied on students who were familiar with refactoring; thus, they had the necessary expertise to state when a refactoring type happened or not. In fact, we also provided a training about all refactoring types in order to help them to recognize them.

Table 3.4 presents the sample sizes of the refactorings manually analyzed by type and the precision obtained to each one. In general, it was observed a high precision for each refactoring type, with a median of 88.36% (excluding rename method). The precision found in all refactoring types are close/inside the standard deviation (7.73). Applying the Grubb outlier test ($alpha$ = 0.05) we could not find any outlier, indicating that no refactoring type is strongly influencing the median precision found. Thus, the results found to the representative sample analyzed represent a key factor to provide reliability to the other results reported in this study.

We also validated the refactoring tactics. For this validation, we conducted three steps. First, we used Eclipse and the eGit plugin[2] to classify a refactoring as root-canal refactoring or floss refactoring. Second, we used a diff

[2] `http://www.eclipse.org/egit/`

Table 3.4: Results of the manual refactoring validation

| **Refactoring Type** | **Population Size** | **Sample Size** | **Precision** |
|---|---|---|---|
| Extract interface | 133 | 99 | 87.88% |
| Extract method | 7,517 | 366 | 80.60% |
| Extract superclass | 342 | 181 | 93.92% |
| Inline method | 1,528 | 307 | 75.57% |
| Move field | 4,356 | 353 | 96.88% |
| Move method | 1,404 | 302 | 88.08% |
| Pull up field | 465 | 211 | 98.58% |
| Pull up method | 629 | 239 | 78.66% |
| Push down field | 78 | 65 | 96.92% |
| Push down method | 114 | 88 | 88.64% |
| Rename method | 12,752 | 373 | 95.17% |

tool to analyze all the changes in the classes modified by the refactoring operations. Third, we analyzed the tool output, searching for a behavioral change. When finding one, we filled a form explaining it, and we classified the change as floss refactoring. When we did not find a behavioral change, we classified it as root canal refactoring. This second validation were conducted by three researchers from our group given their expertise in refactoring.

## 3.2 Preliminary Study

As discussed in the beginning of this very same chapter, we conduced a preliminary study. The reasons behind it is manifold. First, we had to check if, even in smaller projects, refactorings introduce code smells in a non-ignorable frequency. Once this is confirmed, we could execute a thorough investigation to understand the underlying conditions. Second, we had to develop a software pipeline to collect and process code smells and refactorings. A smaller dataset is ideal to tweak and evolve our software pipeline. Third, we had to identify major threats and mitigate them in order to conduct a large-scale study.

This preliminary study [75] comprises of only one research question similar to the $RQ_1$ presented in Section 3.1.1. The refactoring classification scheme presented in Section 2.3 was used to identify scenarios where refactorings introduce or remove code smells. However, only smaller projects were used for this analysis. The complete list of projects used in this preliminary study is presented in Table 3.5. This table present the name, lines of code and number of commits for each project.

After analyzing all versions of these projects, we were able to detect 2,635 refactorings distributed in 7 different refactoring types. In this study, we only

Table 3.5: Projects used during the preliminary study

| **Name** | **LOC** | **Commits** |
|---|---|---|
| altran/Whydah-UserAdminService | 5,588 | 248 |
| alvyxaz/mayhem-and-hell | 23,109 | 144 |
| bublag/confetti | 12,810 | 198 |
| c2nes/ircbot | 7,681 | 101 |
| doanduyhai/Achilles | 86,222 | 935 |
| elasticsearch/elasticsearch-transport-thrift | 3,709 | 111 |
| furio/alfred-mpi | 4,414 | 130 |
| gertvv/drugis-common | 11,180 | 194 |
| hal/ballroom | 11,752 | 548 |
| ikasanEIP/ikasan | 21,2834 | 1,739 |
| iweinzierl/passsafe | 9,678 | 150 |
| jhalterman/lyra | 5,724 | 147 |
| jnape/Dynamic-Collections | 5,686 | 177 |
| localstorm/market-monitor | 2,318 | 124 |
| MarvinBellmann/WhatsUp | 4,014 | 108 |
| mdelapenya/ghprb-plugin | 6,887 | 447 |
| moobid/JARA | 3,348 | 103 |
| OpenConext/OpenConext-cruncher | 2,578 | 106 |
| PhiCode/philib | 16,448 | 880 |
| pusher/pusher-java-client | 5,333 | 249 |
| robertdj20/Containing | 13,323 | 737 |
| rwe17/MediaMagpie | 35,703 | 331 |
| Sen-Word-Builder/Word-Builder | 4,955 | 89 |
| TUBAME/migration-tool | 76,406 | 193 |
| tupilabs/tap4j | 13,823 | 85 |

used the five code smells presented in Table 3.2 instead of using all seventeen types presented in Section 2.2. All refactorings were labeled according to our refactoring classification scheme presented in Section 2.3. In this study, the most common classification detected in our data set was the neutral one. The vast majority of refactorings did not change the number of code smells (95.1% of the times). Negative refactorings represent 2.66% of the cases. Finally, positive refactorings represent 2.24% of the cases. Therefore, observing the dataset, refactorings tend to maintain unaltered the density of code smells. On the other hand, when refactorings affect smelly elements of a program, they slightly introduce more smells rather than reduce them. When we analyze each individual project, the same classification distribution is observed, i.e., neutral refactorings represent the vast majority in all the projects.

Differently from the main study, in the preliminary study we were not sure if we should consider or not the *Rename Method* refactoring. The reasons that motivated us to keep the *Rename Method* in the analysis were manifold.

Table 3.6: Refactoring types and classifications in the preliminary study

| **Refactoring Type** | **Occurrences** | **Applied in Smelly Code** | **Neutral** | **Negative** | **Positive** |
|---|---|---|---|---|---|
| Rename method | 1,491 | 22 (1%) | 1,486 | 4 | 1 |
| Move field | 879 | 726 (82%) | 817 | 32 | 30 |
| Move method | 209 | 168 (80%) | 151 | 33 | 25 |
| Pull up field | 29 | 21 (72%) | 28 | 0 | 1 |
| Pull up method | 21 | 14 (66%) | 20 | 1 | 0 |
| Push down method | 5 | 2 (40%) | 3 | 0 | 2 |
| Push down field | 1 | 1 (100%) | 1 | 0 | 0 |

First, according to Antoniol *et al.* [76], program elements with unstable structural quality tend to also suffer changes in their identifiers. Second, structural change tends to occur in conjunction with *Rename* refactorings [76]. Changes of method and class identifiers seem to be often motivated by structural problems in their implementation [76]. In addition, we considered *Rename Method* because, according to Murphy-Hill *et al.* [1], more than 40% of renames appear as part of a batch within the same commit, *i.e.*, developers often perform rename along with structural refactorings (possibly affecting our list of smells). Thus, we could capture effects of other refactorings by analyzing renames too. On the other hand, we decided to remove the *Rename Method* refactoring from the main study because we considered it as risky. Since this type of refactoring does not involve structural changes, any smell introduction or removal might not be related directly to the refactoring, but to different code changes.

To investigate the impact of refactoring on the existence of smells, We compute the occurrences of positive, negative and neutral refactorings for each type of refactoring (Table 3.6). The goal is to understand the variation of the frequency of positive, negative and neutral refactorings across refactoring types. We computed how many times each type of refactoring was applied in program elements hosting at least one code smell. For each refactoring, we verified whether the refactored elements have smells or not. This factor is captured in the third column (*Applied in Smelly Code*), which shows how many refactorings are related to smelly elements. The following columns present the refactoring classification according to the criteria defined in Section 2.3.

If we compare the data of the second and third columns, we can observe that developers tend to often apply refactorings in smelly elements of a program. Except for the *Rename Method*, from 40% to 82% of the refactorings were applied to smelly elements of a program. The neutral classification was by far the most frequent one for all refactoring types. This result indicates that refactorings are not removing smells even though refactoring was meant to it. In fact, when we consider only the positive and negative refactorings, most refactorings do not remove smells. In other words, even though refactorings often target smelly elements, they often do not reduce the smells present in those elements after refactoring operations. Moreover, negative refactorings occurred as often as positive refactorings.

We can not make any further claims since this study was limited regarding the sample. However, This preliminary study suggests that refactorings tend to neglect the existence of smells, *i.e.*, they often do not remove or introduce smells. Additionally, this study was essential to improve our tooling support and also our study design. For instance, we reasoned if we should maintain the *Rename Method* results in the dataset or not. Also, this study was instrumental for us to came up with new insights and hypothesis. For instance, in this preliminary study we did not consider refactoring-smell patterns, but through the analysis of the initial results, we were able to find new research opportunities. Hence, the main study could benefit from the improved design, which is result of the preliminary study. Furthermore, this was the first study that presented refactorings as capable of introducing new code smells. Definitely, this was the primary motivating factor for the main study, which is thoroughly reported in this chapter. The results for the main study are presented next.

## 3.3 Refactoring and Smells

This section presents and discusses the data used to answer the first research question: *Does refactoring reduce the density of code smells?* The refactoring detection procedure identified 51,461 refactorings. Table 3.7 presents the refactoring types ordered by the number of their occurrences across the projects analyzed. The first column shows each refactoring type followed by the corresponding number of its occurrences (second column) in all projects analyzed.

Table 3.7: The impact of common refactorings types

| **Refactoring Type** | **Occurences** | **Applied in Smelly Code** | **Neutral** | **Negative** | **Positive** |
|---|---|---|---|---|---|
| Extract Method | 7,517 | 6,411 (85.2%) | 2,917 (38.8%) | 3,914 (52.1%) | 686 (9.1%) |
| Move Field | 4,356 | 3,362 (77.1%) | 3,784 (86.9%) | 438 (10.1%) | 134 (3.1%) |
| Inline Method | 1,528 | 1,134 (74.2%) | 732 (47.9%) | 214 (14.0%) | 582 (38.1%) |
| Move Method | 1,404 | 1,049 (74.7%) | 1,008 (71.8%) | 297 (21.2%) | 99 (7.1%) |
| Pull Up Method | 629 | 511 (81.2%) | 430 (68.4%) | 155 (24.6%) | 44 (7.0%) |
| Pull Up Field | 465 | 333 (71.6%) | 338 (72.7%) | 103 (22.2%) | 24 (5.2%) |
| Extract Superclass | 342 | 131 (38.3%) | 89 (26%) | 246 (71.9%) | 7 (2.0%) |
| Extract Interface | 133 | 65 (48.8%) | 11 (8.3%) | 122 (91.7%) | 0 (0%) |
| Push Down Method | 114 | 98 (85.9%) | 76 (66.7%) | 11 (9.6%) | 27 (23.7%) |
| Push Down Field | 78 | 58 (74.3%) | 62 (79.5%) | 13 (16.7%) | 3 (3.8%) |
| **Totals** | 16,566 | 13,152 (79.4%) | 9,447 (57%) | 5,513 (33.3%) | 1,606 (9.7%) |

At first glance, the table already shows that the most common refactoring type is *Extract Method*. This result is particularly interesting since it is similar to what was reported by a previous study, which analyzed refactoring frequencies in other systems [1]. In that study, they reported about *Extract Method* is the most common refactroring type. We can face this similar result as an indicator that our procedure to identify refactoring is appropriated, at least to what concerns the aforementioned related study [1]. Table 3.7 confirms most of the other refactoring types also occur frequently in our sample.

Before addressing $RQ_1$, we first analyze the frequency of refactoring types that touch smelly elements. Thus, we also compute how many times each type of refactoring was *applied in smelly code* (Section 2.2.2). The results are shown in the third column in terms of both absolute number of occurrences and percentages (in brackets). We can observe that developers tend to often apply refactorings in smelly elements of a program. Seven refactoring types have been applied in smelly code elements in more than 70% of the occurrences (Table 3.7). Such result indicates that developers concentrate their efforts to refactor elements with a certain structural degradation. In other words, this result suggests that developers indeed focus on code smells, intentionally or not, to refactor the code. Thus, using the presence of code smells as an indicator of refactoring is not only reasonable but also aligned to how developers apply refactoring in practice.

One could wonder if many elements are tagged as smelly in the analyzed programs, thereby increasing the probability of refactorings often touching smelly elements. Then, we computed the probability of randomly choosing a smelly element in our dataset ($|\text{smelly elements}|/|\text{all elements}|$), which is 0.3%. This low probability shows that, in our dataset, refactorings did not target smelly elements by coincidence. Refactorings indeed tend to concentrate on smelly elements, which were confined to a vast minority of the program elements. This behavior was consistently observed for both root-canal and floss refactorings.

## 3.3.1 Smell-Neutral Refactorings are Common

The three last columns of Table 3.7 present respectively the incidence rate of neutral, negative and positive refactorings. Surprisingly, the neutral classification was the most frequent one for 7 refactoring types, namely *Move Field*, *Inline Method*, *Move Method*, *Pull Up Method*, *Pull Up Field*, *Push Down Method*, and *Push Down Field*. Even though refactorings are frequently applied in smelly elements, they often do not reduce the smells.

As discussed in Section 3.1.2.2, we used a metric-based technique, which also relies on thresholds, to identify code smells. Consequently, different thresholds can lead to different results [44, 74]. Thus, we used two sets of thresholds to avoid bias. The data presented in Table 3.7 was produced with smell detection strategies based on a set of *tight* thresholds. To make sure our findings were not biased by this particular set of thresholds, we have also classified the refactorings using *relaxed* thresholds. Finally, we have also used another set of detection strategies, the same used by Bavota *et al.* [21]. Figure 3.1 shows the general proportion of neutral, positive and negative refactorings using all these three classification methods, labeled as *Tight*, *Relaxed* and *Bavota*.

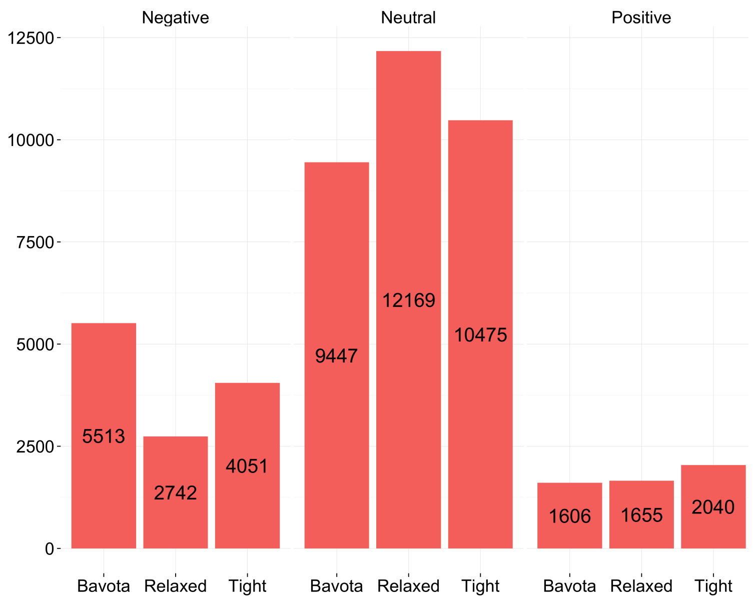


Figure 3.1: Results of the data collection phase

An analysis of Figure 3.1 confirms there is indeed a general trend: independently of the smell detection strategy, neutral refactorings are much more frequent than positive and negative refactorings. When we analyze each individual project, the same classification distribution is observed, *i.e.*, neutral refactorings represent the vast majority in all the projects. In our manual validation (Section 3.1.2.4), we found that 31.5% of the refactorings are root-canal. Even when this tactic is applied, refactorings often do not reduce the density of code smells in the refactored elements. These findings suggest developers need more guidance to remove a code smell once they start to refactor a smelly element, specially when they perform root-canal refactorings.

### 3.3.2 Stinky Refactorings

Surprisingly, 33.3% of the refactorings were found to be *stinky* (Table 3.7); they are related to an increase of smells in the refactored elements. Moreover, when we analyzed the commits performed after the negative refactorings, we also concluded that more than 95% of refactoring-induced smells were not removed afterwards. Only 9.7% of refactorings removed smells, according to Table 3.7. Negative refactorings were more frequent than positive refactorings according to our three classification methods presented in Figure 3.1. Stinky effects are more frequent than positive ones in the context of both root-canal and floss refactorings as well.

Negative refactorings were more frequent than neutral ones in the context of three refactoring types: *Extract Method*, *Extract Superclass* and *Extract Interface*. Interestingly, *Extract Method* is the most frequent type of refactoring (Table 3.7). Section 3.4 discusses different patterns involving this refactoring type. Moreover, the refactorings that involve multiple changes in a class hierarchy, such as *Extract Superclass* and *Extract Interface*, tend to be negative. This fact might indicate developers need more guidance on refactoring class hierarchies even in the context of root-canal refactoring.

These results enable us to answer $RQ_1$: refactorings made by developers in real projects often do not remove code smells. On the contrary, most of the refactorings are neutral or stinky. This observation also prevails if we only consider refactoring types that, according to their description in Fowler's catalog [3], are explicitly associated with specific code smell types addressed in our study. For instance, the mechanics for applying *Move Method*, *Pull Up Method* and *Move Field* refactorings are associated with smells that represented methods or fields that are misplaced. The misplacement of these members are captured by occurrences of either *Feature Envy*, *Divergent Change*, *Shotgun Surgery* or *God Class*.

Our data suggest that most refactorings do not remove smells even in the context of root-canal refactorings. There are possible interpretations of this finding. First, critical design problems in a program may not be related to code smells. If so, this fact may explain why developers either neglect or introduce code smells through refactoring. However, previous studies [5, 9, 34, 50] indicate that design problems are often located in modules containing two or more code smells. Second, similarly to previous studies, we use metrics and thresholds to detect all smells. The proper choice of metrics and thresholds may be sensitive to particular developers [44] and other project-specific factors [74]. As a consequence, our detection of code smells may not reflect what developers truly consider as smells. However, our previous studies involving developers [44, 74] suggest that heuristics used by developers are often not essentially different from smell detection strategies adopted in our study. Our understanding is that refactoring indeed ignores or introduces technical debt in the source code. Finally, it may be the case that our set of studied refactoring types are not among those used by developers to actually remove design problems. Still, it is troublesome that developers introduce smells through refactoring regardless its type and tactic. Therefore, these results lead us to our first finding as follow.

**Finding 1**: Refactoring operations usually do not remove code smells. On the contrary, it is more likely to introduce new ones. Even when developers use the root-canal refactoring tactic, they often degrade the structural quality rather than improve it.

## 3.4 Refactoring-Smell Patterns

In order to address our second research question, we analyzed what are the patterns emerging from the relationship between refactorings and smells. Section 2.4 defines three categories of such patterns, *i.e.*, removal, non-removal and creational patterns. Section 3.4.1 focuses on discussing the removal and non-removal patterns, while Section 3.4.2 discusses the creational (*i.e.*, stinky) patterns.

We will focus on discussing patterns in which more than 15% of the instances of a refactoring type was related to instances of a specific smell type. For these patterns, we inspected all the pattern instances in order to understand what happened in each case. In particular, we also confirmed whether the refactoring was directly related to the removal or introduction of the smell. This was an important step as we had pattern instances occurring in the context of either root-canal refactoring or floss refactoring.

For the non-removal and creational patterns, we also analyzed the lifetime of the prevailing and introduced smells related to the non-removal and creational patterns. We checked if such smells – prevailing or emerging in commits involving one or more refactorings – were either removed or not in subsequent commits. We considered *subsequent commits* all those ones performed until the last commit of each project. Therefore, we were able to identify precisely when a particular code smell was removed. Our goal was to understand whether the refactoring-related smell was (or not) temporarily prevailing in the code because the developer was planning to remove the smell in the next commits.

### 3.4.1 Removal vs. Non-Removal Patterns

Table 3.8 presents the cases of removal and non-removal patterns observed. They are alphabetically ordered by the refactoring type. Each row represents a removal and/or non-removal pattern involving a pair of refactoring type and smell type. The first column shows the refactoring type, followed by the smell type in the second column. The next two columns present for each refactoring type the percentage of its instances related to the removal (fourth column) or prevalence (third column) of the corresponding code smell. Patterns with an incidence strength higher than 15% are shown in bold. The last column presents the percentage of root-canal refactoring for each pattern. For instance, the first row informs that 37.5% of the *Extract Interface* refactorings related to *God Class*, either by non-removal or removal pattern, are root-canal refactoring.

Table 3.8: Removal and non-removal patterns

| Refactoring | Code Smell | Non-Removal | Removal | Root-Canal |
|---|---|---|---|---|
| Extract Interface | God Class | **20.3%** | 2% | 37.5% |
| Extract Method | Divergent Change | **34.6%** | 7% | 25% |
| Extract Method | Feature Envy | **42.6%** | 11% | 28.3% |
| Extract Method | God Class | **48.6%** | 0% | 26.2% |
| Move Field | God Class | **29.4%** | **27%** | 48% |
| Move Method | God Class | **51.3%** | **23%** | 8.0% |
| Pull Up Field | God Class | **44.7%** | 8% | 76.1% |
| Pull Up Method | God Class | **61.3%** | 10% | 2.3% |
| Push Down Field | God Class | **55.7%** | 12% | 57.1% |
| Push Down Method | God Class | **54.3%** | 15% | 22.2% |
| Push Down Method | Lazy Class | 2.6% | **52%** | 11.1% |
| Push Down Method | Refused Bequest | 9.2% | **23%** | 50% |

At a first glance, it is already possible to observe there was a much higher incidence of non-removal patterns than removal ones. The percentages of non-removal patterns (third column) are often higher than their removal counterparts. Thus, we can also conclude there are specific types of refactorings tending to consistently affect a particular type of smell. However, those refactorings more frequently are unsuccessful (non-removal) rather than successful (removal) with respect to that particular smell type. For instance, *Extract Method* refactoring was often targeted at methods hosting a *Feature Envy* smell. However, as expected, most of those *Extract Method* refactorings could not remove this smell. Table 3.8 shows 42.6% (against 11%) of such refactorings touched this smell, but they were not able to eliminate it.

After analyzing these pattern instances, we confirmed that proper action of developers should also include moving (not only extracting) those *Feature*

*Envy* smells as methods to other classes. However, the vast majority of those extracted methods (higher than 95% of its instances) were neither moved to neighbor classes in subsequent commits. In fact, those (11%) of successful *Extract Method* refactorings were performed in conjunction with other method-moving refactorings in the same commit, such as *Move Method*, *Pull Up Method* or *Push Down Method* refactorings.

Another interesting observation is that the *God Class* smell was the most frequent target of removal or non-removal refactorings. In fact, this smell dominates the rows of Table 3.8. Several refactoring types were often related to changes moving out members from *God Class* smells. Two of the refactoring types – namely, *Move Method* (23%) and *Move Field* (27%) refactorings – were significantly successful in contributing to the removal of a *God Class* smell within a commit. However, even for these refactoring types, there was higher incidence of non-removal patterns. Table 3.8 shows 51.3% and 29.4% of *Move Method* and *Move Field* refactorings touched *God Class* smell but were not sufficient to eliminate it, independently if they were part of root-canal or floss refactorings. Those refactorings were often performed in conjunction with other member-moving refactorings in the same commit, but were not sufficient to remove *God Classes.* In 99% of the cases, the prevailing *God Class* smell were not removed in the successive commits either. There were only two refactoring-smell patterns that more predominantly removed (rather than not) the code smell. They were patterns involving the *Push Down Method* refactoring and *Lazy Class* smell (52%) and *Refused Bequest* smell (23%).

We can observe a non-ignorable frequency of root-canal refactorings spread across the patterns in Table 3.8. Even when the root-canal frequency is as high as 50%, the developers are not able to remove the code smell. Since the refactorings belonging to the patterns could be just the first step towards the code smell removal, we computed the code smells' lifetime after the refactoring. In 95% of the cases, the code smells were not removed. This shows that, even when developers refactor purely to improve the code structure (root-canal), they do not succeed on removing the code smells. Hence, these results lead us to our next finding.

**Finding 2**: There are both non-removal and removal patterns. However, the first is way more common than the later. Developers apply specific refactoring types on code affected by specific smells consistently. Still, they are not able to remove them frequently.

### 3.4.2 Creational Patterns

Interesting data also emerged from creational patterns detected in our dataset. We divided these patterns into three groups [3] considering the purpose of the refactoring type: (i) refactorings targeted at improving generalization; (ii) refactorings responsible for moving features between objects; and (iii) refactorings targeted at restructuring members of a class. The following subsections respectively present and discuss creational patterns involving refactorings in these groups. Table 3.9 presents all creational patterns found with the same structure presented in Table 3.8.

Table 3.9: Creational patterns

| Refactoring | Code Smell | Creational | Root-Canal |
|---|---|---|---|
| Extract Method | Divergent Change | 40.5% | 24.7% |
| Extract Method | Feature Envy | 63.8% | 32.1% |
| Extract Superclass | Lazy Class | 33.2% | 17.8% |
| Extract Superclass | Refused Bequest | 20.3% | 0% |
| Extract Superclass | Spec. Generality | 68.3% | 0% |
| Move Method | Complex Class | 15% | 0% |
| Move Method | God Class | 35% | 17.6% |
| Move Method | Lazy Class | 16% | 16% |
| Pull Up Field | God Class | 61.2% | 2% |
| Pull Up Field | Spec. Generality | 34.1% | 90.2% |
| Pull Up Method | God Class | 28.3% | 2.9% |
| Pull Up Method | Spaghetti Code | 23.2% | 0% |

#### 3.4.2.1 Generalization Patterns

Refactorings dealing with generalization were often related to the creation of *God Class* and *Speculative Generality* smells. We can observe in Table 3.9 that *Pull Up Method*, and *Pull Up Field* refactorings are related to the creation of *God Class* smells in 28%, and 61% of the cases, respectively. *Extract Superclass* refactoring creates the *Speculative Generality* smell in 68% of the cases, while 34% of the *Pull Up Field* refactoring instances introduce this same smell in the target superclass. What is more troublesome was the fact that more than 95% of such introduced smells were not removed in successive refactorings.

A typical example of generalization-related creational pattern can be illustrated by the case involving the *DefaultProjectListener* class from the

Xerces project (commit *002901b*). The *DefaultProjectListener* class is the default implementation of a listener that emulates the old ant listener notifications. *Extract Superclass* refactoring was applied on *DefaultProjectListener* class, thereby creating the *AbstractProjectListener* class from it. However, the new abstract class did not seem to justify the refactoring. There was only one class that extended *AbstractProjectListener* class, i.e. the *DefaultProjectListener* class itself. Thus, the refactoring created the *AbstractProjectListener* class with a *Speculative Generality* smell. Moreover, this refactoring had another negative consequence on the affected classes as it introduced another code smell. The *DefaultProjectListener* class overrides all the methods defined on the *AbstractProjectListener* class. Consequently, the *DefaultProjectListener* class became affected by a *Refused Bequest* smell. One of the reasons for this problem is that all the bodies of the methods defined on the *AbstractProjectListener* class are empty; they do not have any implementation. Ideally, the *AbstractProjectListener* abstract class should have been instead defined as an interface. Moreover, all these smells were not removed in successive commits, thereby affecting other listener subclasses created latter. Therefore, in this example, the *Extract Superclass* refactoring is responsible for creating an instance of a generalization-related creational pattern and propagating a smelly structure to other classes.

#### 3.4.2.2 Feature-Moving Patterns

Refactorings aiming at moving features between objects were also part of our catalog of detected creational patterns. *Move Method* refactorings were related to the creation of three types of smells. This refactoring created *God Class*, *Complex Class*, and *Lazy Class* smells in 35%, 15%, and 16% of the cases, respectively. Interestingly, this type of refactoring was amongst the most common ones in a previous study [1]. When analyzing all these pattern instances, we confirmed that developers were consistently creating smells through *Move Method* refactorings in the target classes (*i.e.*, those receiving the moved methods) without removing those smells in the source classes. Again, the vast majority of these introduced smells (more than 98%) prevailed in the successive commits. This observation shows that tooling support should warn developers about the risks related to such recurring creational patterns.

A typical case of creational feature-moving pattern can be illustrated by refactoring changes affecting two classes from the ArgoUML project. The *generateMessageNumber* method was moved from the *GeneratorDisplay* class to the *MessageNotationUml* class. Before the refactoring, the *GeneratorDisplay* class had three types of code smells, namely *God Class*, *Complex Class* and *Refused Bequest* smells. On the other hand, the *MessageNotationUml* class had only one code smell: *Refused Bequest*. After the refactoring, the *MessageNotationUml* class received the other three types of code smells that were affecting the *GeneratorDisplay* class. However, the *GeneratorDisplay* class continued having the three types of code smells. That is, in addition to introducing code smells in the target class, *Move Method* refactoring did not remove the code smells from the source class. To make matters worse, *Move Method* refactoring also introduced a fifth type of code smell that was not affecting any one of the both classes before. It introduced a *Spaghetti Code* smell since the moved method interacts, through a method call, with an existing method of the *MessageNotationUml* class that was long (in terms of LOC). This *Move Method* refactoring instance is a critical one since it is responsible for creating two out of three relevant code smells (*God Class* and *Complex Class* smells), and it is also responsible to introduce the *Spaghetti Code* smell.

#### 3.4.2.3 Method Extraction Patterns

In the last category, we only found creational patterns involving the *Extract Method* refactoring. This refactoring type was often related to the creation of two types of smells: *Divergent Change* smell in 41% of the cases, and *Feature Envy* smell in 64% of the cases. However, when we analyzed these pattern instances, we observed that most of them occurred in the context of: (i) floss refactorings, or (ii) composite root-canal refactorings. Therefore, *Extract Method* refactoring was often not the only factor potentially contributing to the emergence of those code smells. Still, the high incidence of such creational patterns may warn developers that *Extract Method* refactorings should be often followed by *Move Method* refactorings in order to eliminate possibly prevailing *Feature Envy* or *Divergent Change* smells.

The *FixCRLF* class from the Apache Ant project had the *Complex Class* smell. Also, the *execute* method from this class had two code smells, namely *Feature Envy* and *Long Method* smells. This method had two functionalities: executing a scanning task on a source code folder and processing files found in the folder. Through the *Extracted Method* refactoring, the *execute* method was split into a second method called *processFile*. After the refactoring, the *execute* method had only the *Feature Envy* smell while the *processFile* method kept both smells: *Feature Envy* and *Long Method* smells. However, it was also introduced a *Divergent Change* smell in the *processFile* method.

Furthermore, it was introduced another code smell in the *FixCRLF* class. After the refactoring, it also had a *Spaghetti Code* smell. In this example, the *Extracted Method* refactoring was responsible for creating an instance of a method extraction pattern. Also, this refactoring contributed to introduce a code smell at class level, *Spaghetti Code*, which did not exist in the class before the refactoring, and together with the *Complex Class* smell, they can decrease the reusability of the system.

Table 3.9 presents the percentage of root-canal refactorings for each creational pattern. Considering all instances involved in those patterns, 26.5% are root-canal refactorings. This is an alarming rate. Developers introduce smells when refactoring even when they are performing solely structural-improvement activities. To make the matter worse, this behavior occurs consistently between specific refactoring-smell pairs. This non-tolerable rate presented indicates developers might need proper support during refactoring to avoid structural degradation even when they, clearly, want to improve the structure via root-canal refactoring. Hence, the results reported in this section lead us to our next finding as follow.

**Finding 3**: Creational patterns are surprisingly frequent. Some refactoring types tend to create new code smells of particular types consistently.

## 3.5 Sequence of Refactorings

We found that refactorings usually do not remove code smells (Finding 1). This result is intriguing since most refactorings are applied to smelly elements; thus, one might expect that these refactorings should have a positive impact. However, they are not removing smells even though it is meant to it. This result lead us to at leas two further discussions, both related to the need of applying multiple refactoring operations. First one, developers need to apply a sequence of refactorings because one could not remove the code smell completely. The second one, developers are not applying refactoring properly; thus, they need to apply a sequence of refactorings to achieve their purpose.

Unfortunately, we cannot delve into these discussions because we do not know the developers' motivation to refactoring. Either way, we expect that in some cases developers have to apply a sequence of refactorings. For instance, to remove a *God Class* smell, a developer could need to apply multiple *Extract Class* refactorings, which each operation would extract a responsibility. Even code smells considered simpler would need to apply multiple refactorings to remove it completely. For instance, sometimes only part of a method suffers from *Feature Envy*; in this case, a developer can use *Extract Method* to get the "envious" code, and Move Method to move it to the appropriate class. Indeed, a study showed that 40% of the times developers apply two or more refactoring operations in the same code element [1]. In these cases, we should not consider the effect of each single refactoring in the presence of smells but the effect of the batch. If we consider the effect of each single refactoring, we are neglecting how refactoring is applied in the practice. For instance, if we have a batch in which the developer had to apply two refactorings to remove the smell completely. In this case, we have an example of a (batch) refactoring with a positive impact in the system, since it removed the code smell. Therefore, we also should consider the impact of batch refactoring in the existence of smells. Hence, we will find out if developer successfully removed the *God Class* or ended up introducing other smells. In summary we have to investigate the impact of batch refactorings on the existence of smells.

Actually, to investigate if batch refactoring introduces or removes smells is even more needed since a recent study showed that a single refactoring may not suffice to fully remove certain types of smells [45]. In this study, the authors looked at the internal quality attributes such as cohesion, coupling, complexity and size, which are capture by metrics used to detect smells. In fact, these quality attributes represent symptoms of code smells. For instance, a *God Class* is likely to have a high coupling and a low cohesion. Consequently, metrics for coupling and cohesion are used to detect *God Class*. According to the authors, 65% of the refactoring operations improve their related internal quality attributes and the remaining 35% operations keep the quality attributes unaffected. This result indicates that developers need to apply a sequence of refactoring to completely remove the smell. A single refactoring may improve a particular metric that captures one of these quality attributes, but it does not improve all the metrics (or symptoms) associated with the smell. Consequently, developers would need to apply another refactoring operation. In summary we have to investigate the impact of batch refactorings on the existence of smells, which is conducted in Chapter 5.

## 3.6 Threats to Validity

This section discusses the study limitations based on the threats to the study validity, presenting the measures took to mitigate these threats.

### 3.6.1 Internal Validity

The data collection using the Refactoring Miner represents a threat to internal validity because it may find false positives. To minimize this threat and check the tool's precision, we randomly select a sample of 2,584 refactorings and manually validate them. Section 3.1.2.4 presents the procedure used to estimate the precision of this tool in our dataset. We observed a high precision for each refactoring type, with a median of 88.36%. The precision found in all refactoring types are close to the standard deviation (7.73). By applying the Grubb outlier test (alpha=0.05), we could not find any outlier, indicating that no refactoring type is strongly influencing the median precision found. Thus, the results found in the sample analyzed represent a key factor to provide confidence in the results reported in this work.

We could not reach the developers to ask their intentions (root-canal or floss) in all refactorings detected. Therefore, we performed a manual validation analysis to check whether each refactoring instance was part of a root-canal or floss refactoring. This manual validation also represents a threat to internal validity. To mitigate this threat, part of the sample was validated by two researches. The third researcher solved cases of conflict.

The answers to both research questions rely on the code smells. Thus, different thresholds can lead to completely distinct results. Therefore, choices of thresholds can pose a threat to this study. Thus, three sets of thresholds and rules are used to mitigate this menace: *tight* [5], *relaxed*, and *Bavota* [17]. These three smell detection strategies allow us to derive refactoring classifications based on three different sets of code smell instances, mitigating the threat of the thresholds choice. Following such approach, we could not find any project from the analyzed sample influencing the results in a significant way. The set of code smells types can be also considered a threat to validity. However, we consider very common smell types, not to mention we also used Bavota *et al.* [17] technique, allowing us to improve the coverage of different code smell types.

### 3.6.2 External Validity

As several empirical studies involving software systems, the lack of representative results is also a possible threat to the external validity. For example, the number and domains of programs used in the study may not be enough to generalize the results. In order to further reducing the impact of this threat, we considered a large dataset with programs from a wide range of domains. Nevertheless, all the programs used in the study constitute both programs widely used in other empirical studies and popular Java projects on GitHub. We did not observe diverging results across different program domains. Moreover, we also included in the sample the three projects used by Bavota *et al.* [17], so that we could contrast our findings with theirs (Section 2.5).

## 3.7 Related Work

Section 2.5 presents many studies somehow related to this thesis. In this section, we offer a comparison between the study presented in this chapter to some studies shown in Section 2.5. To do so, we divided this section into groups of studies with the same thematic.

### 3.7.1 Elements Touched by Refactoring

Bavota *et al.* [17] mined the evolution history of 3 Java open source projects to investigate if refactorings occur on code elements that certain indicators suggest a need for refactoring. Their considered indicators include structural quality metrics and the presence of smells. They also measure the effectiveness of refactorings regarding their ability to remove smells. According to their results, quality metrics do not show a clear relationship with refactoring and 42% of the refactorings are applied on smelly elements, in which only 7% of them remove smells. Different from Bavota *et al.*, our results indicate that most of the refactorings (80%) are performed in elements with code smells, in which 9.7% of them remove smells and 33.3% induce new smells.

The procedure that we followed may explain why our results are different from the one presented by Bavota *et al.*. We collected refactorings between commits while they collected refactorings using only the projects' major versions. Usually, between two major versions, developers perform significant changes in the source code structure. Therefore, they probably missed refactorings when they followed this procedure since refactorings might be hidden or unidentifiable. In our study, we mitigate this threat by collecting the refactorings between consecutive commits. In summary, our study improves several aspects of the study reported by Bavota *et al.*. First, we analyzed 23 projects while they analyzed only three. Second, we collected refactorings in consecutive commits. Third, we used the Refactoring Miner refactoring detection tool [28] which have a good precision rather than the well-known Ref-Finder's low precision [55]. In addition, we evaluated if and when refactorings are stinky by introducing new smells in the context of both root-canal and floss refactoring. We have also characterized refactoring-smell patterns.

### 3.7.2
### Motivations to Refactor

Our data showed that most of refactorings (80%) touch smelly elements. Even though the specific motivation of the developers is unknown, we actually observed a similar behavior when developers apply both root-canal and floss refactoring. Mainly in the former case, one would expect developers explicitly intend to improve code structure. Regarding developers' motivation, Silva *et al.* [15] investigated the reasons that drive developers to refactor their code. Their results indicate that fixing a bug or changing the requirements, such as feature additions, mainly drives refactorings. Their results show that the refactored code may contain code smells, although developers did not mention it explicitly as the intention to refactor. On the other hand, the study of Yamashita and Moonen [10] reports that developers often consider smells as critical.

### 3.7.3
### Benefits to Refactor

Kim et al. [29] conducted a three-folded investigation at Microsoft about refactoring through a survey, interviews, and data analysis of the Windows version history. Their results indicate that the refactoring in practice seems to differ from the rigorous definition of behavior-preservation transformation found in the literature [3]. This result provides us an additional support to investigate floss refactorings since developers also perform non-behavior-preservation transformations while refactoring. The survey participants also reported the benefits they have observed from refactoring. The two most cited benefits were improved readability and maintainability. Although the participants did not mention code smells, the presence of code smells affects negatively the two benefits they claim they want to achieve [10, 50, 77].

### 3.7.4
### Refactoring Recommendation Systems

Silva *et al.* [15] mentioned that future studies on refactoring recommendation systems should refocus from code-smell-oriented to the maintenance-task-oriented solutions. Nevertheless, we think that such refocus needs more reflection. Instead, a refactoring recommendation system should employ a hybrid solution, in which both code-smell-oriented and maintenance-task-oriented solutions are combined. Thus, a envisaged recommendation system can support developers to conclude their maintenance tasks in a way that smells are also removed or not introduced. Our set of non-removal and stinky patterns (Section 3.4.2) can be used to help constructing such a recommendation system.

### 3.7.5 Introduction of Code Smells

Tufano *et al.* [51] investigated the circumstances that led to the introduction of smells, not specifically in the context of software refactoring. The authors analyzed issues and tags associated with commits that introduced smells on open source projects. Their results indicate that most of code smells are introduced during enhancement activities (between 60% and 66%). They also found that between 4% and 11% of the smell-introducing commits were tagged as refactoring. However, our study goes beyond: our findings indicate that stinky refactorings are frequent: refactorings are related to the introduction of code smells in 33.3% of the cases. We also characterized and quantified typical refactoring-smell patterns, and observed that certain stinky patterns are very frequent.

### 3.7.6 Relation Among Code Smells, Refactoring and Other Software Aspects

Khomh *et al.* [40] investigated the relation between 29 smells and changes occurring in classes from two software projects. Their results showed that classes with code smells are more likely to change than classes without a smell. This result may help us to understand why most of the refactoring instances touch smelly elements. Fujiwara *et al.* [13] and Ratzinger *et al.* [11] studied whether refactoring reduces the probability of software defects. The authors find that an increase in refactoring has a positive interference on software quality. Results show that number of defects in the target period decreases if more refactorings are applied. While the authors correlate refactorings with bug fixes and the interference of such changes on software defects, we are correlating smells with refactorings.

### 3.7.7 Negative refactorings

We conducted a preliminary study [75] that analyzes how often the commonly-used refactoring types affect the density of 5 types of smells along the version histories of 25 projects. Our findings are based on the analysis of 2,635 refactorings distributed in 11 different types. Surprisingly, 95.1% of refactorings are neutrals. Only 2.24% of refactoring were positive, and 2.66% negatives. Our new results show that negative refactorings are much more frequent. This discrepancy between the studies can be explained by the nature of the projects. In the preliminary study, we considered only small projects with fewer developers.

## 3.8 Summary

We conducted a study aiming to understand the relationship between refactorings and code smells in 23 projects. First, we have observed that although refactorings touch smelly elements, they are often smell-neutral. Second, stinky refactorings occur more often than positive refactorings. Stinky refactorings were also surprisingly frequent in root-canal refactorings, *i.e.*, when developers are solely focused on improving the program structure. These findings suggest developers need more guidance to fully remove a code smell once they restructure a smelly element.

In order to better guide developers, we have investigated which recurring refactoring-smell pairs tend to produce stinky, neutral or positive effects. We achieved this goal by revealing and characterizing removal, non-removal and creational patterns. We found a wide range of creational and non-removal patterns, which were much more frequent than positive patterns. *Extract Method* is a refactoring type frequently involved in both stinky and non-removal patterns. Moreover, we decomposed creational patterns in three groups. The first group included refactorings dealing with generalization: they were often related to the creation of *God Class* and *Speculative Generality* smells. The second group represents feature-moving refactorings, which induced the creation of *God Class*, *Complex Class*, and *Lazy Class* smells. Finally, the last group comprises the *Extract Method* refactorings, which were related to the creation of *Divergent Change* and *Feature Envy* smells.

The aforementioned findings can help practitioners and tool engineers. Practitioners using refactoring are now better informed of when they may typically introduce neutral or stinky refactorings in their programs. Moreover, a refactoring assistance tool can be built in order to: (i) detect when developers performed refactoring in a commit, and (ii) depending on the refactoring characteristics (*e.g.*, occurrence of a root-canal refactoring), immediately produce warnings or recommendations for the developer. For example, a refactoring assistant could warn developers of an emerging *God Class* (related to a stinky refactoring) and suggest her to move the class member to a more appropriate class.

Some studies already reported refactoring operations are frequently performed in batches [1, 16]. However, in both studies presented in this chapter, we only considered refactorings in isolation, *i.e.*, single refactorings. Hence, we might had been evaluating only parts of a whole work being performed by the developers. For instance, a negative refactoring could be transformed in a positive one if the developer applied more refactorings in a batch manner. This batch refactoring phenomenon motivated further studies, which we need to perform to better understand the impact of refactoring on the existence of code smells. Before conducting theses studies, the next chapter introduces concepts related to batch refactoring.

# 4
# Batch Refactoring: Characterization and Synthesis

Chapter 2 presents the conceptual framework used for referring to single refactoring, code smells, and single refactoring classification. At the end of the Chapter 3, we introduced the need for studying batch refactoring operations. Like Chapter 2, this chapter also outlines the terminology adopted throughout this thesis. However, in this chapter we extend the conceptual framework presented before. Here, we focus on formally defining aspects related purely to batch refactoring.

## 4.1
## Refactorings Flock Together

Due to the diversity of the purposes behind refactoring, researchers have been trying to understand how and why developers perform refactoring [1, 15, 16]. These studies report the most common refactoring types applied by developers [1], the typical reasons underlying several refactoring types [15], and how developers use tools that help to refactor [16].

Among other findings, these studies briefly shed light upon the existence of *batch refactoring*, *i.e.*, when developers apply a sequence of single refactoring operations. For instance, [1] reported that 40% of the times developers apply two or more refactoring operations in the same code element. Indeed, even to remove some code smells, developers may need to apply multiple refactorings. For instance, a developer may need to apply multiple *Extract Class* refactorings to remove a *God Class*. Sometimes, only a single refactoring is not enough to remove a smell completely, as presented in [45]. Thus, we also need to investigate if batch refactoring introduces or removes smells. In this work, we formally define batch refactoring as follow.

### 4.1.1
### Batch Refactoring

In our context, a batch refactoring comprises two or more refactorings performed in a sequence that have been applied by the same developer [1]. In a formal way, a sequence of single refactoring operations is considered a batch refactoring (or, simply, a *batch*) if some constraints are satisfied. First, a batch must have more than one single refactoring. Second, all refactorings in a batch must have been performed by a single developer. Third, if $b = [r_1, r_2, \cdots, r_n]$ is a batch of size $n$, then $[r_1, r_2, \cdots, r_n]$ is a list of refactoring operations ordered chronologically.

We have restricted the batch to those sequences of refactorings applied by the same developer since it is likely that the same developer starts and finishes the batch by himself. We defined this criterion because we consider that a batch comprises a sequence of operations towards a goal defined by a single developer. Thus, if a developer starts a batch, he probably has a goal set. Another developer hardly would share the same goal in order to interleave refactorings with the sequence of refactorings performed by another developer (e.g. in the same commit). Consequently, a second developer may rarely interfere during the batch being performed by the first one. In fact, our recent analysis shows that a sequence of refactorings in a module by a developer is rarely intertwined with a sequence of refactorings performed by another developer [78]. Moreover, when two batch sequences of different developers affect the same module, the "distance" is usually higher than one month or dozens of commits.

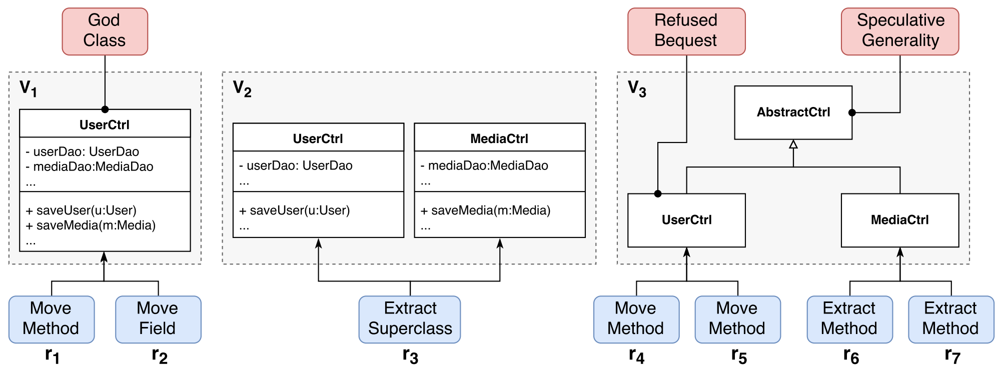


Figure 4.1: Batch refactoring example

For instance, let us revisit the example presented in Chapter 2, which is presented again[1] in Figure 4.1. The developer performed two refactorings in the *UserCtrl* class in the first version. In this way, the batch $b_1 = [r_1, r_2]$. The second possible batch occurs in the class *MediaCtrl* in version $v_3$. The developer applied two *Extract Method* refactorings. In this way, $b_2 = [r_6, r_7]$. Both batches differ in some characteristics. For instance, $b_1$ has two different refactoring types, while $b_2$ has only one. A batch may have additional characteristics, such as its timespan, heterogeneity, and scope, as defined in the next sections.

[1]Since this example is several pages back in the text, we show the same picture again in Figure 4.1 for the reader's convenience

### 4.1.2 Batch Type

A batch refactoring can have a type in the same way that a single refactoring has one. The type of a given batch $b$ is defined as the ordered list of refactoring types that it contains. Formally, $type(b) = \cup_{i=1}^{n} type(r_i)$. So, the outcome of the function *type* is (i): if it is applied in a single refactoring, it merely returns the refactoring type, and (ii) if it is applied in a batch, it returns the ordered list of refactoring types that the batch contains. Hence, $type(b_1) = [Move\ Method, Move\ Field]$. To know the batch type, *i.e.*, all the refactoring types that compose the batch are important. For example, with this information, we can identify which are the refactoring types that happen in a batch that often introduces or removes smells. This information can be used for awaring developers to either avoid or pursuit these batch types in a given context.

### 4.1.3 Batch Timespan

As discussed in Section 2.1.1, we need to compare two subsequent versions of a system to identify a refactoring. In this context, all the refactorings in a batch can happen in the same version, or they can happen in different subsequent versions. For instance, $r_1$, $r_2$ and $r_3$ refactorings happened in $v_1$, $v_2$ and $v_3$ versions, respectively. The batch timespan indicates if the batch is either *cross-version* or *single-version*.

This batch characteristic is useful to understand if the developers usually apply batches in a single version of the software or if the batches are spread into several subsequent versions. In this way, the *timespan* characteristic is useful to understand the behavior of the developers during batch refactorings. We can infer if they usually work focused in a batch and finish it in a single version or if they usually interleave batch refactorings with different changes in a code during several versions of the software.

To identify the batch timespan, let us define the function $version(r)$ as the version where the refactoring $r$ was performed. In this way, we can say that a batch $b = [r_1, r_2, \cdots, r_n]$ is *cross-version* if and only if $|version(r_1) \cup \cdots \cup version(r_n)| > 1$. Hence, if a batch is *cross-version*, then the refactorings belonging to it occur in more than one version. Similarly, if $|version(r_1) \cup \cdots \cup version(r_n)| = 1$, then $b$ is *single-version*. In our example, both batches $b_1$ and $b_2$ are *single-version*.

### 4.1.4 Batch Heterogeneity

As a batch is a sequence of refactorings, all the refactorings in the batch can have the same type or not, which we define as batch heterogeneity. The batch $b = [r_1, r_2, \cdots, r_n]$ is *heterogeneous* if and only if $|type(r_1) \cup type(r_2) \cdots \cup type(r_n)| > 1$. In this way, if a batch is *heterogeneous*, then we can find more than one type of refactoring on it. Similarly, if $|type(r_1) \cup type(r_2) \cdots \cup type(r_n)| = 1$, then we can say the batch is *homogeneous*. For instance, in our example, we can observe the two types of batches regarding heterogeneity. The $b_1$ batch is *heterogeneous*, while $b_2$ is *homogeneous*.

The heterogeneity is also useful to understand how developers behave themselves during the application of multiple single refactorings. By observing this characteristic, we can understand if they tend to apply multiple refactoring types in a batch or if they tend to apply only refactorings of the same type. Since several batch refactorings presented by Fowler [3] are heterogeneous, it is reasonable to think developers apply heterogeneous batch refactorings in practice. However, little is known about this batch characteristic in real scenarios.

### 4.1.5 Batch Scope

Similar to refactorings, batches also have a scope. The batch scope is the set of code elements related to the batch. An intuitive way of defining batch scope can use the notion of refactoring scope. One might naturally say the union of all refactoring scopes involved in a batch determines the batch scope, but this is not necessarily true in all scenarios. In fact, the way the batch is created determines its scope. The refactoring scopes do not necessarily define the batch scope. The Section 4.2 presents different ways of identifying batches, and, for each one, we present how the batch scope is determined.

The batch scope is a crucial characteristic for our study. It determines the blast radius of the batches, i.e., the portion of the code changed by them. By observing how the scope structurally changed after a given batch, we are able to classify the batch and work towards answering our research questions 3 and 4.

## 4.2 Batch Synthesis Heuristics

According to the previous definition, a batch is a sequence of single refactorings. In other words, a batch is a group composed of single refactorings.

The process of grouping refactorings to create batches is defined as *Batch Synthesis*. Also, the way the batches are synthesized is open-ended, i.e., different developers can have different views of how to create a batch. For instance, one might say a batch is composed only of refactorings performed in a single version, while others might say the version is not so relevant, but just the scope of the refactorings is. Since this study aims at understanding the impact of batches on the structural quality of the systems, we hypothesize the way the *Batch Synthesis* occurs can impact the results of this study. Therefore, we define three batch synthesis heuristics in the following sections. These heuristics empower us with the capability of analyzing the batches from three different perspectives.

Studies that report the existence of batches [1, 16] had access to real-time data of refactoring activities, i.e., they observed developers performing refactorings *in loco*. In this way, they could visually check whether a developer is consciously applying a batch or not. For instance, Murphy-Hill *et al.* [1] considered that a sequence of refactorings form a batch if the developer did not spend more than 60 seconds between each single refactoring. Still, they only considered refactorings performed through the IDE options. In our study, we focus on the analysis of the source code repositories. Our data is less granular, but our data sample is way larger. Moreover, we believe that the criterion of "60 seconds" is way to restrictive. Thus, we have defined and applied three heuristics that follow three different criteria.

To detect batches, we used three different heuristics created based on observations reported by other authors. For instance, Fowler [3] presents examples of batch refactorings used to remove code smells in some different ways. Multiple *Extract Methods* can be performed in a single method in order to remove a *Long Method*. In this case, all refactorings were performed in a single code element. This behavior led us to propose the *element-based* synthesis heuristic. Still, according to Fowler, the developer might want to move some of the extracted methods to a different class, involving more than one code element. Although the elements are different, they are structurally related (same scope). This kind of scenario led us to the *range-based* heuristic.

Finally, Murphy-Hill *et al.* [1] considered that a sequence of refactorings form a batch if the developer did not spend more than 60 seconds between every single refactoring, independent on the structural relation of the refactored code elements. Since it is reasonable to think that all refactorings performed in a time-window of 60 seconds would be in the same software version, we then proposed the *version-based* heuristic. The next sections present all the heuristics.

### 4.2.1 Version-Based Heuristic

The *version-based* batch synthesis heuristic considers only the versions as a criterion to group refactorings. It means that all refactorings performed in a single version are grouped together to synthesize a batch. In this way, at most one batch might be synthesized by version. Formally, a given batch $b = [r_1, r_2, \cdots, r_n]$ is synthesized by the *version-based* heuristic if and only if $|version(r_1) \cup version(r_2) \cdots \cup version(r_n)| = 1$. For instance, consider again the Figure 2.1. Hence, $H(s_1) = [r_1, \cdots, r_7]$. Now, let $B_v(h)$ be the function that implements the *version-based* synthesis heuristic over a particular refactoring history $h$. Thus, $B_v(H(s_1)) = \{b_a[r_1, r_2], b_b[r_4, r_5, r_6, r_7]\}$. Therefore, in this example, the *version-based* synthesis heuristic produce two batches: $b_a$ and $b_b$, since in each batch all refactorings were performed in the same version.

**Scope** In this heuristic, the batch scope is determined by the set of elements affected by the refactorings belonging to the batch. In this way, $scope(b_a) = \{UserCtrl\}$, and $scope(b_b) = \{UserCtrl, MediaCtrl\}$.

### 4.2.2 Element-Based Heuristic

The *element-based* batch synthesis heuristic considers only a single element as heuristic to group refactorings, even if the refactorings were applied in different versions. It worth mentioning that, as presented before, all refactorings in a batch must have been performed by a single developer. Formally, a given batch $b = [r_1, r_2, \cdots, r_n]$ is synthesized by the *element-based* heuristic if and only if there is an element $e$ such as $e \in scope(r_i)\ \forall r_i \in b$. For instance, let $B_e(h)$ be the function that implements the *element-based* synthesis heuristic over a particular refactoring history $h$. So, $B_e(H(s_1)) = \{b_c[r_1, r_2, r_3, r_4, r_5], b_d[r_3, r_6, r_7]\}$. Thus, this heuristic synthesize two batches by analyzing $H(s_1)$. The first one, $b_c$, is a batch according to this heuristic because $[r_1, r_2, r_3, r_4, r_5]$ affected the same element *UserCtrl*. The second batch, $b_d$, is synthesized because all refactorings belonging to it were applied in the $MediaCtrl$ class.

**Scope** In this heuristic, the batch scope is determined by the element used to synthesize the batches. In this way, $scope(b_c) = \{UserCtrl\}$, and $scope(b_d) = \{MediaCtrl\}$.

### 4.2.3 Range-Based Heuristic

Both *version* and *element-based* heuristics consider a single factor to synthesize batches. In the first case, only the version is considered, while in the second, a single element is considered. The *range-based* synthesis heuristic considers the notion of refactoring scope to synthesize the batches. In this heuristic, the batch starts with an arbitrary refactoring $r_a$. A second refactoring $r_b$ is part of the same batch if and only if it was performed by the same developer of $r_a$ and $\exists e \in \mathit{scope}(r_b)$ such as $e \in \mathit{scope}(r_a)$. A possible third refactoring $r_c$ will be added to the batch if the same developer performed it and $\exists e \in \mathit{scope}(r_c)$ such as $e \in \mathit{scope}(r_a)$ or $e \in \mathit{scope}(r_b)$. This process continues until all refactorings in a particular history are explored. The Algorithm 1 describes how the *range-based* heuristic synthesizes batches for a given system $s$, *i.e.*, this algorithm defines $B_s$.

**Algorithm 1** Range-based synthesis algorithm

```
Input: a system $s$
Output: a set of batches
1: $B \leftarrow \emptyset$
2: for all $r \in H(s)$ do
3:     $matches \leftarrow \emptyset$
4:     for all $b \in B$ do
5:         if $\exists e \in scope(r)$ such as $e \in scope(b)$ then
6:             $matches \leftarrow matches \cup b$
7:         end if
8:     end for
9:     $B \leftarrow B - matches$
10:    $batch \leftarrow \{r\}$
11:    if $|matches| > 1$ then
12:        $batch \leftarrow merge(matches) \cup \{r\}$
13:    else if $|matches| = 1$ then
14:        $batch \leftarrow matches_1 \cup \{r\}$
15:    end if
16:    $B \leftarrow B \cup batch$
17: end for
18: $cleanup(B)$
19: return $B$
```

The algorithm starts by iterating over all refactorings of the history of the system $s$ (line 2). For each refactoring $r$, the algorithm identifies all already-synthesized batches where at least one element of $\mathit{scope}(r)$ belongs to the batch scope (lines 4–8). The batch where the refactoring $r$ will reside depends on how many matches we found. If multiple matches were found, then we merge all of then into one (line 12). If only one match is found, then $r$ will be added to this batch (line 14). If no matches were found, then a new batch is created containing only $r$ (line 10). The brand new batch is added to the set of already-synthesized batches (line 16). At the end, this might lead to batches containing only one refactoring. Therefore, before finishing the algorithm, we call the *cleanup* function to delete such batches (line 18).

The $r_1$ and $r_2$ refactorings in Figure 2.1 moved elements from *UserCtrl* to *MediaCtrl* classes. Hence, $\mathit{scope}(r_1) = \mathit{scope}(r_2) = \{UserCtrl, MediaCtrl\}$. The batch synthesis in this example starts with $r_1$. Since $r_2$ was applied in one element of $\mathit{scope}(r_1)$, then the batch grows bigger and turns into $[r_1, r_2]$. The $r_3$ refactoring affects elements of $\mathit{scope}(r_1)$, then the batch is now $[r_1, r_2, r_3]$. The same reasoning can be used for the remaining refactorings, so the batch synthesis produce the batch $[r_1, r_2, r_3, r_4, r_5, r_6, r_7]$.

**Scope** In this heuristic, the batch scope is determined by union of the scopes of all refactorings belonging to it. In this way, the scope of a batch synthesized by the *range-based* heuristic is defined as $\cup_{i=1}^{n} \mathit{scope}(r_i)$.

## 4.3 Batch Classification

In addition to studies that try to understand how and why refactoring happens, there are studies that investigate the impact of the refactorings in the source code. Bavota *et al.* [17] investigate the capability of refactoring on removing code smells, while Cedrim et al. [79] studied also the capability of smell introduction. A common limitation of such studies is that they only consider single refactorings, disregarding whether they are part of a batch or not. Therefore, the conclusion of these studies might be inaccurate as developers often apply refactoring operations together rather than in isolation [1]. Our study aims at filling this gap in the literature, since each batch may exert certain impact on the program structural quality.

To observe how batch affects the presence of smells, this chapter extends the refactoring classification scheme presented in Section 2.3. In the previous scheme, the refactorings are classified according to their impact on the presence of code smells. Each refactoring is divided into positive, negative, and neutral. A given refactoring is said to be positive if the number of code smells before the refactoring is greater than after. On the contrary, a refactoring is negative if the number of code smells after is greater than before. If there is no change in the number of code smells, then we classify the refactoring as neutral. The batch classification proposed in this chapter uses the same notion of positive, negative, and neutral categories. However, the notion to compute the number of smells before and after is different from the previous classification scheme, which is explained as follow.

## 4.3.1 Code Smells in the Batch Scope

In order to investigate the impact of batch refactoring on the existence of code smells, we need to look at the refactored elements. In other words, we need to verify if there is code smells in the batch scope. For this purpose, we can adapt the already defined function *ScopeSmells* (Section 2.2.1) to the context of batch refactoring. To understand how this adaptation works, assume that exists a batch $b = [r_1, r_2, \cdots, r_n]$. To identify the smells in its scope, we changed the *ScopeSmells* to interact over each refactoring within a batch, returning the smells in the scope of each refactoring. Thus, the function instead of receiving a single refactoring as defined in Section 2.2.1, it receives a list of refactorings, which composes the batch. In this way, $ScopeSmells(b, v)$ returns all code smells existing in the scope of the batch $b$ considering the version $v$ as follow.

$$ScopeSmells(b, v) = \bigcup_{i=1}^{|scope(b)|} smells(e_i, v) \tag{4-1}$$

The *ScopeSmells* function defined above gives us a way to obtain all code smells existing in the batch scope for a particular version. Once we defined a pair of versions, we can use *ScopeSmells* to observe whether the number of smells decreased or not between both versions. Since $b = [r_1, r_2, \cdots, r_n]$ is a list of refactoring operations ordered chronologically, then $version(r_1)$ is the version when the batch begins, and $version(r_n)$ when it ends. Therefore, $ScopeSmells(b, version(r_1))$ represents all existing smells in the version when the batch started. Similarly, $ScopeSmells(b, version(r_n))$ represents the smells in the version when the batch ended. Hence, we can now define the batch classification scheme.

## 4.3.2 Positive, Neutral and Negative Batches

Using the data returned by the functions defined before, it is possible to classify a batch by looking how it interferes in existing code smells. We rely on the same classification defined in Section 2.3. Thus, a batch is classified as a *positive* one if it reduces the number of code smells. Conversely, it is classified as a *negative* one if it increases the number of smells. Otherwise, it is classified as *neutral* if it neither increases nor decreases the number of smells. To perform such classification, we rely on the *ScopeSmells* function. For instance, suppose $ScopeSmells(b, version(r_1)) = x$, and $ScopeSmells(b, version(r_n)) = y$. Depending on $x$ and $y$, it is possible to classify $b$. If $x > y$, $b$ reduced the number of smells on *scope*$(b)$ and, because of that, $b$ is considered a *positive batch*. Otherwise, if $x < y$, $b$ increased the number of smells on *scope*$(b)$; thus, $b$ is a *negative batch*. When $x = y$, $b$ is a *neutral batch*.

To better illustrate this classification, consider the batch $b_a = [r_1, r_2]$ synthesized by using the *version-based* heuristic. Before $b_a$, there was one *God Class* code smell. After applying $b_a$, the code smell was removed and no other code smell emerged. Hence, $b_a$ is a positive batch. Now, let us consider only the versions $v_1$ and $v_2$ and the *range-based* heuristic. In such scenario, this heuristic would synthesize the batch $b_e = [r_1, r_2, r_3]$. Before this batch, only one smell exists: *God Class*. After, we can observe two: *Speculative Generality* and *Refused Bequest*. Therefore, $b_e$ is an example of negative batch.

## 4.4 Smell-Batch Patterns

Section 2.4 presents three types of patterns between single refactorings and code smells. After concluding the study reported in Chapter 3, we were able to detect and report several refactoring-smell patterns (Section 3.4). Those results led us to conjecture about the existence of patterns between batches and smells. This conjecture is intuitive because, even for simple code smells, Fowler *et al.* [3] defined a sequence of single refactorings that would be able to remove them. Indeed, developers may often need to apply several refactorings to fully remove certain types of smells [45] Therefore, not only it is realistic to reason about specific batch types that often remove particular code smells but it is also necessary. Such necessity arises since we need to understand how developers apply (multiple) refactorings before providing them support.

Unfortunately, there is no study in the literature that reports batch types that often introduce code smells. Before studying those scenarios, we extend the refactoring-smell patterns definition in the next sections. For this definition, let us consider $B_h = \{b_1, b_2, \cdots, b_n\}$ be the set of all detected batches by the heuristic $h$ after analyzing the set $S$. Thus, $B_{bt}$ is the subset of $B$ of batches of the type $bt$. The set $B^{+}_{bt,cs}$ is the $B$ subset composed of batches of the type $bt$ that added code smells of type $cs$ in any element belonging to the batch scope, while $B^{-}_{bt,cs}$ is the $B$ subset that removed code smells of type $cs$. Given this initial notation, we now can define the two kinds of smell-batch patterns.

### 4.4.1 Batch Creational Patterns

The first smell-batch pattern, namely *batch creational pattern*, represent cases where the batch refactoring introduced code smells. In other words, a batch creational pattern occurs when a specific batch type involves code transformations that often introduces a specific code smell. Similarly to the definition of creational batches to single refactorings, we also define the batch creational pattern concept as a threshold-based rule. If $|B^{+}_{bt,cs}|/|B_{bt}| \geq \sigma$, it is possible to affirm that there is a batch creational pattern between $bt$ and $cs$. This kind of pattern captures scenarios where developers apply a batch refactoring and, somehow, end up creating at least one new code smell. Thus, batch creational patterns represent cases of stinky batch refactorings. These patterns are those that developers should be aware during refactoring since they have a negative impact in the software system.

### 4.4.2 Batch Removal Patterns

The second smell-batch pattern is the one that remove code smells after the refactorings: *batch removal pattern*. The definition of for this batch also lies in a threshold-based rule. If $|B^{-}_{bt,cs}|/|B_{bt}| \geq \sigma$, we can affirm that there is a removal pattern between $bt$ and $cs$. It means that developers consistently removes instances of $cs$ when performing batch refactorings of the type $bt$. This kind of batch-smell pattern is the one that has a positive impact in the software system. Even though any study has investigated the batch removal patterns, we expect that these patterns comprise the ones reported by Fowler *et al.* [3]. By formalizing this kind of pattern in practice, we are able to observe if the refactoring mechanics proposed by Fowler *et al.* [3] occurs in real scenarios during code smells removal.

## 4.5 Towards the Investigation of Batch Refactoring

In this chapter, we presented and discussed concepts related to batch refactoring. These concepts are an extension of the conceptual framework presented in Chapter 2. Our first discussion was regarding the fact that refactorings flock together. Indeed, we noticed in Chapter 3 that single refactorings usually do not remove code smells. Such results led us to wonder why these code transformations are not reducing the density of code smells. During this discussion and relying on the literature [1, 3, 45], we concluded that often a single refactoring may not suffice in removing a code smell completely. Thus, we should also investigate if, after other refactorings operations, these code smells were successfully removed. To conduct such an investigation, we need to investigate the sequences of refactorings applied to the smelly element. This sequence of refactorings is what we named *batch refactoring*. Since batch refactoring is commonly applied by developers [1], we noticed the need to investigate their impact on the existence of smells.

However, before conducting such an investigation, which is described in the next chapter, we had to define heuristics to identify batches. We defined three heuristics: *version-based*, *element-based* and *range-based*. After defining these heuristics, we are capable of identifying the batches that occur in the software projects. Consequently, we can investigate their impact on the software systems. In order to investigate the impact of batch refactoring on the existence of code smells, we adapted the classification scheme from Section 2.3 for the context of batch refactoring. Thus, a batch can be either *positive*, *negative* or *neutral* if it decreases, increases or not affects the number of code smells, respectively. Based on this classification, we can further investigate the impact of batch refactoring on the existence of code smells. We can also find the patterns that lead to the removal or introduction of code smells. This investigation is discussed in the next chapter.

# 5
# Investigating the Impact of Batches on Smells

In our journey towards the understanding of the refactoring practice, we first studied how single refactorings change the density of code smells (Chapter 3). In this study, we collected 51,461 refactorings spread into 113,306 commits. Each refactoring collected went through a classification process (Section 2.3) according to its impact on the code smells. Thus, this study led us to the first evidence that refactoring operations usually do not remove code smells. On the contrary, it is more likely to introduce new ones. Even when developers use the root-canal refactoring tactic, they often degrade the structural quality rather than improve it. Since, by definition, refactoring is a way of improving the code structural quality, the findings of this study are, somehow, surprising. However, the findings were revealed by considering only single refactorings.

Although important, studies that consider solely single refactorings are capable of only scratching the knowledge surface of the refactoring practice. In order to better understand how refactorings affect the existence of code smells, we have to dig deeper and consider batch refactorings. In practice, 40% of the times developers apply two or more refactoring operations in the same code element [1]. Moreover, according to Fowler [3], it might be necessary to perform multiple refactorings to remove a code smell instance. Considering these observations, it is likely that previous studies provide a limited view about the impact of refactorings on smells.

In order to address such limitation, we conducted a study with two purposes. First, we investigate what characterizes a batch. Thus, we analyzed the data collected in our previous studies to understand what constitutes a batch. Based on this analysis, we created a heuristic to automatically detect them. Later, we detected batch refactorings in 48 software projects. After this, we observed how code smells were affected by each batch refactoring found. By analyzing the impact of batch refactorings, we check if batch refactorings affect code smells differently from single refactorings. Therefore, we can contribute to filling the gap present in the literature about the impact of batch refactorings on code smells. In this vein, we found the first evidence that batch refactorings often do not reduce the density of code smells. In fact, most of the batches are also neutral or negative.

In the context of single refactorings (Chapter 3), we present several refactoring-smell patterns. Similar patterns might also emerge when batch refactorings are taken into consideration. In this context, we call them batch-smell patterns (Section 4.4). The study presented in this chapter also comprises the identification of such patterns. Their existence might reveal what are the common refactorings used by developers when they are removing – or even introducing – code smells. We were able to identify several batch-smell patterns, and some of them are presented in this chapter.

This chapter is organized as follows: Section 5.1 provides the goal and research questions that guide this study. Section 5.1.2 presents the study planning. In Section 5.2 we present the results regarding the impact of batch refactorings on code smells, while in Section 5.3 we present some of the batch-smell patterns found. We present the threats to validity in Section 5.4, while the related work is in Section 5.5. We present a summary and concluding remarks of this chapter is in Section 5.6.

## 5.1 Study Settings

This section presents the settings of our study. In Section 5.1.1, we present the research questions that we intend to answer. In Section 5.1.2, we present the study design followed to answer our questions.

### 5.1.1 Goal and Research Questions

Chapter 3 presents our first study towards the understanding of how refactorings affect code smells. However, we only considered single refactorings during its execution. As discussed before, developers might need more than one refactoring in a single code element to remove a code smell [3], i.e., they might need a batch refactoring. As mentioned earlier, studies suggest that batch refactorings are common [1]. Nonetheless, batch refactorings are still poorly understood, mainly when their impact on code smells are taken into consideration. In this context, the goal of this study is stated as follows:

**Goal:** Understand how batch refactorings affect code smells.

In order to classify a single refactoring as negative, neutral or positive, we consider the number of smells before and immediately after the refactoring operation (Section 2.3). Therefore, what happens with the code smell later does not interfere in the refactoring classification. Let us assume that a particular developer applies a negative single refactoring. A few commits later, the developer applies a new refactoring in the same code element and removed the code smell. In this way, if we consider both refactorings as a batch, we can say this batch is neutral, since the first refactoring introduced a smell and the second one, removed it. Clearly, if we consider batch refactorings, the classification results might differ from what was reported in Chapter 3. Therefore, our study aims at addressing the following research question:

**RQ$_3$.** Does batch refactoring impact the density of code smells?

A similar research question was answered in Chapter 3. We investigated whether single refactorings reduce the density of code smells or not. Surprisingly, we found that single refactorings are way more prone to introduce than remove code smells. On the dataset we used, single refactorings introduced code smells in 33% of the cases, while they remove smells only in 9.7% of the times. However, developers apply batch refactorings often [1]. Since we first studied only single refactorings, we might have analyzed only fragments of the developer work. For instance, let us revisit the example presented in Figure 4.1 (Chapter 4). Analyzing $r_1$ in isolation, we can say it is a positive refactoring because it removed the *God Class* code smell. However, when analyzing the batch $[r_1, r_2, r_3]$, we observe the developer work led to a negative batch. Despite being essential to analyze single refactorings, analyzing its impact individually can be a minimalist analysis of how refactorings impact code smells. However, there is a necessity of analyzing the impact of batches on code smells, as expressed by RQ$_3$.

We address this question by relying on the classification of each batch detected in real projects. This procedure enables us to compute how frequent each batch classification occurs across the projects. First, we detected instances of refactorings and code smells. Then, all batches were classified according to Section 4.3. Let $p$ the number of batches classified as positive; $n$ the number of negative batches; and $k$ representing the number of neutral batches. If $n > p$ and $n > k$, we can state that the application of batches are likely increasing the number of code smells. Otherwise, if $p > n$ and $p > k$, the answer to our research question is **yes**, batches tend to remove code smells. Another possible case is when $k > p$ and $k > n$. In this scenario, batches would tend to neither introduce nor remove code smells.

It is also important to understand and distinguish the impact of specific batch types on code smells. Fowler et al [3] presented a catalog of batch refactorings that can be used to remove code smells. For example, suppose that there is a method affected by the *Feature Envy* code smell. In this case, Fowler recommends to apply a batch refactoring composed of *Extract Method* and *Move Method*, in this order. Thus, some types of batches might consistently remove specific smell types (or fail to do so). In this particular case, one would expect each instance of the batch *Extract Method* and *Move Method* would remove a *Feature Envy* if the latest was present in the refactored code.

On the other hand, we presented some unexpected relations between some types of refactorings and types of smell (Section 3.4). For example, our data suggest that *Move Method* refactorings can, often, create *God Class* instances. We observed that a common reason was due to the fact that the developer moves the method to an inadequate target class. In other words, another class should receive the method being moved from its source class. There were also cases where the destination class received new methods from multiple *Move Method* refactorings, even from different source classes. In this case, the creation of such code smell might be related to the multiple application of *Move Method* refactorings. Hence, we hypothesize that unexpected relations like that might happen between batch and smell types. It might be the case that some specific batch types can, frequently, introduce specific smell types across software projects. Section 4.4 defined categories of patterns between types of batches and smells. These patterns are the focus of our next research question.

**RQ$_4$.** What are the patterns governing batches and code smells?

By answering RQ$_4$, we might be able to reveal batch refactorings used by developers not only to remove, but also to inadvertently introduce code smells. We detect removal and creational patterns by analyzing the impact of batch types on smells located in the batch scope. The knowledge about creational patterns make developers informed about the risks of introducing certain smells along batch refactoring.

### 5.1.2 Study Phases

This section presents the phases of the study design. Section 5.1.2.1 describes the dataset we used for batch refactoring synthesis and classification. After this, Section 5.1.2.2 describes the last two data collection procedures: batch synthesis and classification.

#### 5.1.2.1 Phase 1: Dataset Acquisition

Section 3.1.2 presented the phases of the first study we conducted to answer RQ$_1$ and RQ$_2$. The first two phases are *Selection of Software Projects*, and *Smell and Refactoring Detection*. Since the study presented in this current chapter aims at analyzing the impact of batch refactorings on smells, the same dataset used in the first study (Section 3.1.2) was used here. One of the goals is to allow a comparison of the findings originated from the single refactorings study to the ones reported in this current chapter. It is worth mentioning we described two different sets of software projects in Chapter 3. The first dataset was used in the preliminary study (Section 3.2), and the second one in the main study (Section 3.1.2). For the batch refactoring study we merged both datasets into a single dataset. Therefore, in order to answer $RQ_3$ and $RQ_4$ we rely on refactorings and code smells collected from 48 different projects (Tables 3.1, and 3.5). Therefore, all refactorings and code smells previously collected are used in this study of batch refactoring.

#### 5.1.2.2 Phase 2: Synthesis and Classification of Batches

As described in Section 5.1.2.1, we used a single dataset composed by the projects presented in Tables 3.1 and 3.5 to study batch refactorings. In this way, we already have the refactoring history (Section 2.1.2) of each project presented in these tables. Moreover, as presented in Section 4.2, all three heuristics for synthesizing batch refactorings requires a refactoring history as input. Figure 5.1 illustrates how the batch synthesis occurs in this study. First, all refactoring histories are collected (single refactorings study). After this, each refactoring history is submitted to the synthesis algorithms described in section 4.2. In this way, for each project considered, three sets of batches are collected: (i) element-based batches, (ii) range-based batches, and (iii) version-based batches.

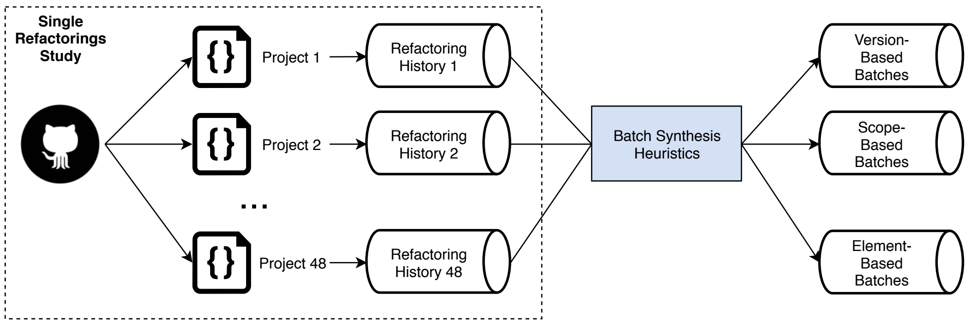


Figure 5.1: Batch synthesis procedure

Once all batches are synthesized, we can start analyzing them. The first step towards answering $RQ_3$ is the classification procedure. As described in Section 4.3, we can classify each batch according to its impact on the code smells. In this way, each synthesized batch is then submitted to the batch classification scheme. After this classification, it is possible to quantify how frequent batches are classified as positive, negative, and neutral.

The study presented in Chapter 3 also presents the single refactorings classified as positive, negative, and neutral. Since we follow a similar classification scheme for batches, we are able to contrast both results. This will give us evidence about how batch refactorings can affect code smells and how they differ from single refactorings in this matter.

## 5.2 Batch Refactoring and Code Smells

This section presents and discusses the data used to answer $RQ_3$. First, we present the single refactoring data (Section 5.2.1). We follow the discussion presenting the synthesized batches and also the batches classification (Sections 5.2.2 and 5.2.3, respectively).

### 5.2.1 Single Refactorings

The refactoring detection procedure identified 51,461 single refactorings. Table 5.1 presents the refactoring types ordered by the number of their occurrences across the projects analyzed. The first column shows each refactoring type followed by the corresponding number of its occurrences (second column) in all projects analyzed. The third column presents the number of projects where the refactoring type was observed. For instance, the *Rename Method* refactoring was observed in all projects, while *Move Field*, appeared in 38 projects. The most common refactoring type is *Move Field*, similarly to previous studies that analyzed refactoring frequencies in other systems [1, 79]. As we can see, all refactoring types were observed in multiple projects, varying from 16 (*Push Down Field*) to 48 (*Rename Method*). This high diversity of projects give us confidence that there is not one or two projects biasing our results. Several projects contributed to the data we present here onward.

Table 5.1 includes rename refactorings in two rows. However, one might wonder why we removed the *Rename Method* and *Rename Class* refactoring types from the single refactoring study, but we kept them in the batch refactoring one. Such refactoring types have no close relationship with any code smell addressed in our study, i.e., the code change needed to perform them is not capable of affecting these code smells. Hence, it was not interesting to study their impact on code smells when analyzing them as single refactorings. However, renames are very popular refactorings [1, 16] and can, often, be

Table 5.1: Single refactorings detected by type

| Refactoring Type | Quantity | Projects | Applied in Smelly Code |
|---|---|---|---|
| Move Field | 15,306 | 38 | 13,743 (89.7%) |
| Rename Method | 15,215 | 48 | 7,178 (47.1%) |
| Extract Method | 9,569 | 47 | 7,551 (78.9%) |
| Inline Method | 3,428 | 39 | 1,307 (38.1%) |
| Move Class | 2,339 | 24 | 978 (41.8%) |
| Move Method | 1,909 | 34 | 1,267 (66.3%) |
| Pull Up Method | 1,171 | 24 | 940 (80.2%) |
| Rename Class | 927 | 24 | 326 (35.1%) |
| Pull Up Field | 757 | 21 | 516 (68.1%) |
| Extract Superclass | 410 | 25 | 136 (33.1%) |
| Extract Interface | 185 | 25 | 82 (44.3%) |
| Push Down Method | 149 | 17 | 120 (80.5%) |
| Push Down Field | 96 | 16 | 64 (66.6%) |
| Push Down Field | 51,461 | | 34,208 (66.5%) |

interleaved with other structurally-relevant refactorings in a batch. In this sense, we keep renames in the batch refactoring study.

The last column of Table 5.1 reports how often instances of a certain refactoring type "touch" at least one smelly element, i.e., we present how many times each refactoring type was *applied in smelly code* (Section 2.2.2). The results are shown in terms of both absolute number of occurrences and percentages (in brackets). For example, 89.7% of the *Move Field* refactorings are applied to program elements containing one or more code smells. We can observe that developers tend to often apply refactorings in smelly elements of a program, as presented in Section 3.3.

The last row of Table 5.1 presents the total number of refactorings. As we can see, 66.7% of the collected refactorings are applied in smelly code. This number, per se, shows that refactorings often target smelly elements. As aforementioned, the table also contains two refactoring types not directly related to the code smells considered in our study, which is *Rename Method* and *Rename Class*. If we disregard both refactoring types, the percentage of refactorings applied in smelly elements becomes 75.6%. This new value is similar to the one reported in Section 3.3 (79.4%). The percentage we found is again higher than the 42% reported in a previous study [17], indicating that developers usually apply refactoring in code elements suffering from structural problems.

This high percentage of refactorings affecting smelly code is particularly relevant to our study. First, it indicates that developers (consciously or not) can be attracted by code smells when they decide to start either a root-canal or floss refactoring. Second, it shows that there is room for structural improvement in the refactored code, i.e., there is possibility of having positive refactorings if the proper transformations are performed by a developer. If most refactorings were performed in smell-free code, the possibility of existing positive refactorings would be tiny as such transformations were most likely only serving to achieve non-structural improvements. In this hypothetical scenario, there was no smell to remove; then, the refactorings could only be neutral or negative. In this way, the values presented in Table 5.1 shows we have a diverse dataset to study the impact of refactorings on smells, empowering us with the capability of answering $RQ_3$.

## 5.2.2 Synthesized Batches

This section presents descriptive data about the synthesized batches. First, we show how many and how long are the batches (Section 5.2.2.1). After this, we present and discuss the other batch characteristics: homogeneity and timespan (Section 5.2.2.2).

### 5.2.2.1 Quantity and Size of Batches

All single refactorings must be processed to synthesize batches (Section 5.1.2.2). In this study, we used the refactoring histories (Section 2.1.2) from 48 projects to synthesize batches considering three different heuristics: version-based, element-based, and range-based. Table 5.2 presents data regarding the number of refactorings belonging to batches. For each heuristic, we first present how many batches were synthesized (second column).

Table 5.2: Batch size by heuristic

| | | Single Ref. | Size | | | | |
|---|---|---|---|---|---|---|---|
| **Heuristic** | **Quantity** | **in Batches** | **Average** | **Std. Dev.** | **Max** | **Min** | **Median** |
| Element-based | 12,636 | 28,394 (54%) | 3.9 | 6.6 | 333 | 2 | 2 |
| Range-based | 3,730 | 28,883 (55%) | 7.7 | 62.2 | 2,556 | 2 | 2 |
| Version-based | 11,545 | 47,218 (91%) | 8.0 | 44.4 | 2,562 | 2 | 3 |

A batch refactoring consists of a list of two or more refactorings. Therefore, after applying the batch synthesis heuristics, a given single refactoring will be either isolate or part of a batch refactoring. Hence, an alternative way of thinking about our dataset is to split it into two separate sets of refactorings: (i) interlinked refactorings – refactorings that are part of at least one batch; (ii) isolated refactorings – do not belong to any batch. In this vein, the third column of Table 5.2 presents the count of the interlinked refactorings for each heuristic. For instance, we synthesized 12,636 batches via the element-based heuristic, totaling 28,394 interlinked refactorings, which represents 54% of the total of single refactorings belonging to our dataset.

When considering the element-based and the range-based heuristics, 54% and 55% of the refactorings are interlinked through a batch, respectively. These numbers are not far to the ones reported by previous studies [1, 16], which state that 40% of the refactorings are performed in batches. This similarity might indicate that these heuristics capture the way the developers applied batch refactorings during these studies. However, 91% of the refactorings belong to batches in the version-based heuristic. It means that developers, when refactoring, apply at least two refactorings in the same commit, even if the refactorings are applied to structurally-unrelated code elements. This might occur, for instance, in cases where developers are refactoring code clones scattered over different parts of the system. In any case, these numbers show that researchers should not study only single refactorings, but they should also consider the batches, which may comprise several interlinked transformations. Therefore, these data lead us to our first finding in this chapter:

**Finding 4**: Developers perform batch refactoring more often than they perform single refactoring.

Our first finding in this chapter is fundamental for the next steps of the batch refactoring study. Since developers perform batch refactoring more often than they perform single refactoring, there is a possibility of getting different results for $RQ_3$ and $RQ_4$ if compared to the results reported in Chapter 3. If batch refactorings were extremely rare in our dataset, answering $RQ_3$ and $RQ_4$ would not be so relevant. Since we found the opposite, we can carry on our investigation.

Still in Table 5.2, we present data regarding the size of the synthesized batches. We present different characteristics of this variable: (i) the average (third column); (ii) the standard deviation; (iii) the maximum and the minimum value observed (fifth and sixth columns); and (iv) the median (last column). As we can see, the values differ a lot among the heuristics. The average size of batches synthesized by the element-based heuristics is 3.9, while 8 is the one for version-based synthesized batches. This variation is expected due to the nature of the synthesis algorithms. For instance, in the element-based heuristic, we only group together the refactorings applied to the same element. On the other hand, the version-based heuristic attaches refactorings to a batch independently from the code element where they were applied; we just consider the commit where they were applied. In this way, we can say the condition to form version-based batches is looser if compared to the one used by the element-based heuristic. Hence, it is expected to observe version-based batches bigger than element-based batches.

By definition, the range-based heuristic is a generalization of the element-based one (Section 4.2.3). In order to visualize that, let us assume a hypothetical scenario where a developer is refactoring the class $A$. She first moves the methods $m_1$ and $m_2$ from class $A$ to class $B$. After this, she extracts part of $m_1$ to a new method in class $B$. If we consider the element-based heuristic, the first two *Move Methods* would form a batch. On the other hand, all three refactorings would be part of a batch if the range-based heuristic is used. The batch scope of the element-based heuristic is fixed (always a single class or method), while the scope of the range-based heuristic is organic – it grows for each new refactoring performed. Therefore, we expected to observe bigger batches in the range-based heuristic if compared to the version-based ones.

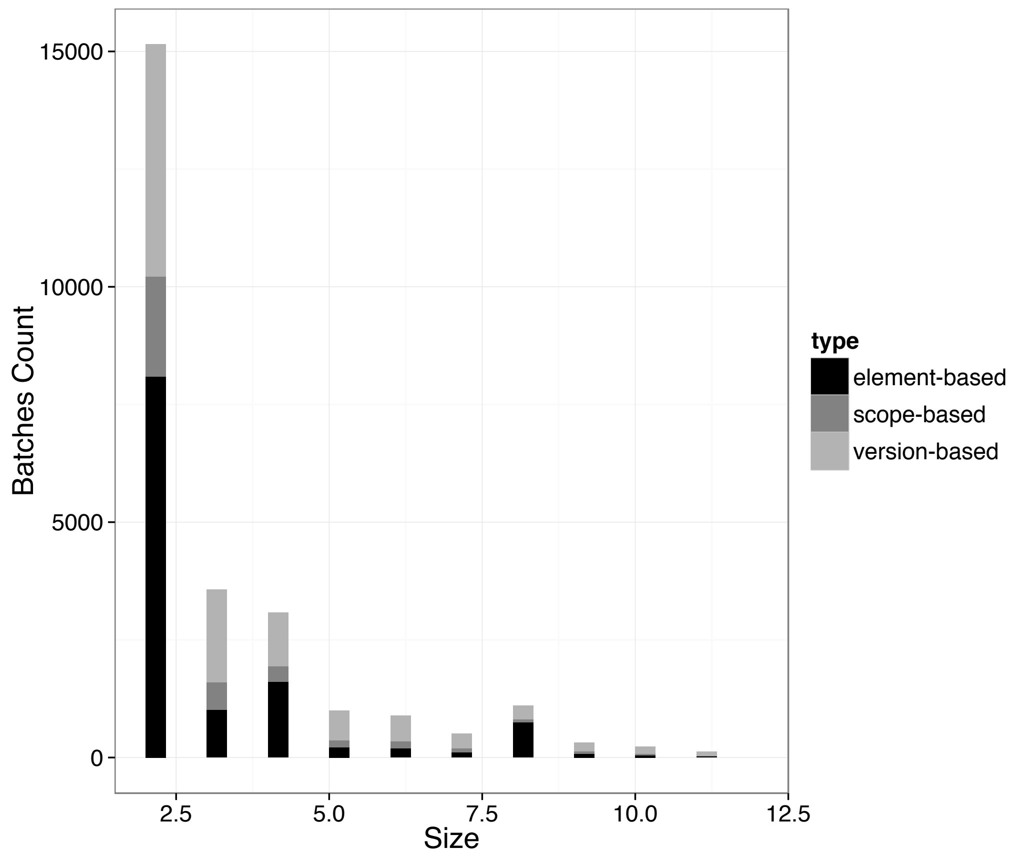


Figure 5.2: Batch size distribution

Figure 5.2 presents how the batch sizes are distributed by each heuristic. As we can see, most batches (independent on the heuristics) is composed of 5 or fewer refactorings. The frequency of batches bigger than that starts to decrease, mainly when we reach the size 10. Actually, in our dataset, we found extremely big batches, as depicted in Table 5.2 (max column). However, these batches are rare, and one might say they are outliers in our dataset. Since most of the batches are small, as shown by Figure 5.2, then most of the results here presented will be directly derived from them, i.e., small batches are the main responsible for our findings. This is an expected result since for removing several code smell types, only a few refactorings are enough [3].

#### 5.2.2.2 Heterogeneity and Timespan of Batches

Table 5.3 presents the results concerning the *timespan* and *heterogeneity* characteristics of batches. The table provides the interrelation between these two characteristics and allows us to see at a glance the proportion of batch refactorings that are either heterogeneous or homogeneous and cross-version or single-version. For each heuristic, the table presents the distribution of batches among such characteristics.

Table 5.3: Timespan and heterogeneity characteristics

| | **Timespan** | | **Heterogeneity** | |
|---|---|---|---|---|
| **Heuristic** | **Single-Version** | **Cross-Version** | **Homogeneous** | **Heterogeneous** |
| Element-based | 9,094 | 3,542 | 11,107 | 1,529 |
| Range-based | 3,486 | 244 | 2,875 | 855 |
| Version-based | 11,545 | 0 | 6,484 | 5,061 |

As we can observe, developers tend to limit the batches in a single commit. This can be an indication that they perform all refactorings they intend to perform at once, i.e., without splitting the task into multiple commits. This is an important result if we think about recommendations. Let us assume we want to develop a recommendation system to suggest batch refactorings aiming at removing code smells. This system can use the leverage of the data presented in Table 5.3 to recommend refactoring that can be done in a single commit, in this way it would mimic the most common behavior of the developers.

Although most of the batches are single-version, a non-ignorable amount of batches are cross-version when we consider the element-based heuristic. If we combine this fact with the one that most of the element-based batches have less than 5 refactorings, we can say that developers apply small batches in different commits in the same elements. Even though they do several cross-commit batches according to the element-based heuristic, most of them are single commit.

Regarding heterogeneity, we can say most of the batches are homogeneous. Both element-based and version-based heuristics produce more than 70% of the batches homogeneously. It means that developers apply the same type of refactoring when restructuring related code elements most of the times. The highest incidence of heterogeneous batches occur in the version-based batches, but this can be explained due to the nature of the version-based batches. According to this heuristic, any refactoring performed in a given commit is part of a batch, even if these refactorings are applied in structurally unrelated elements. As we presented, if we consider related code elements, the batches performed on them tend to be homogeneous if we look to each batch separately. Let us assume the developer applied two *Move Methods* in the class $A$, and two *Move Fields* in the class $B$ in the same commit. According to the element-based heuristic, we would have two homogeneous batches. However, since all refactorings were performed in the same commit, the version-based heuristic would produce a four-sized heterogeneous batch.

In summary, we can say that most of the batch refactorings occur in a single version rather than crosscutting multiple versions. Furthermore, independently from the heuristic, most of the batches are homogeneous rather than heterogeneous. Still, even though most of the batches are single-version and homogeneous, we found a non-ignorable frequency of heterogeneous or cross-version batches. Previous work [1, 22] ignores the fact that developers indeed apply a sequence of types of refactoring along multiple commits in the software history. Hence, after analyzing different batch refactoring characteristics, our results lead us to our next finding:

**Finding 5**: Even though homogeneous and single-version batches are more frequent than its counterparts, heterogeneous and cross-version batches occur with a non-ignorable frequency, which is overlooked by previous research.

This finding shed a light upon the so far unrevealed characteristics of batches, and its impact on our study is significant. In our single refactoring study (Chapter 3), the impact of refactorings on smells were all collected in a single-version fashion, since we only observed the impact of single refactorings. Although cross-version batches are not the majority, they might have a significant impact on the batch classification (Section 4.3). For instance, let us assume there is a tendency of cross-version batches to be positive. The amount of cross-version batches is enough to impact the classification distribution towards a higher percentage of positive refactorings. In this way, this finding is one more evidence that the classification results of batches might differ from the ones reported on Chapter 3.

### 5.2.3 Most Batches are Neutral

Section 4.3 presents a batch classification scheme according to the impact of batches on code smells, so all synthesized batches are classified according to it. Therefore, the three last columns of Table 5.4 present respectively the incidence rate of positive, neutral and negative refactorings. Similarly to what was reported in Chapter 3, the neutral classification was again the most frequent one for all heuristics. Even though refactorings are frequently applied in smelly elements, they often do not impact the amount of prevailing smells.

Table 5.4: Batches classification by heuristic

| Heuristic | Positive | Neutral | Negative |
|---|---|---|---|
| Element-based | 751 (5,9%) | 11,264 (89,1%) | 621 (4,9%) |
| Range-based | 542 (14,5%) | 2,020 (54,2%) | 1,168 (31,3%) |
| Version-based | 1,653 (14,3%) | 6,019 (52,1%) | 3,873 (33,5%) |

Section 3.3 presents a similar classification scheme considering only single refactorings. In the dataset we used before, 57% of the refactorings were classified as neutral. Also, we report that 9.7% were positive, and 33.3% negative. In this previous work, we considered single refactorings and, also, found a high incidence of neutral refactorings. However, since most of the refactorings belong to batches (Section 5.2.2), most of these single refactorings used to report the percentages are interlinked through a batch. Some might say this fact might introduce bias in the classification percentage we reported, since we are evaluating the impact of the same refactoring twice – one time in the single refactoring study, and again in the batch study. In this way, we computed the classification frequencies for all isolated refactorings, i.e., the ones that are not part of a batch. Considering only the isolated refactorings subset, we obtained the following percentages: 35.7% of negatives, 55.8% of neutrals, and 8.5% of positives. Again, even in this subset, the percentages are almost the same, giving us confidence about the findings.

When analyzing the refactorings in batches (Table 5.4), a similar behavior is observed. Independently from the heuristic, most of the batches are neutral. This can be happened due to our classification scheme that only considers the number of smells to classify the batches. In order to illustrate the concern we had with this, consider a scenario where before the batch $b$ the scope contained two code smells, being one *Feature Envy* and one *Message Chain*. After $b$ being performed, the developer ended up with also two smells: one *Feature Envy*, and one *God Class*. In this case, our classification scheme would say *b* is neutral. However, a *God Class* would be often considered worse than a *Message Chain*. Hence, in this hypothetical case, it would not be fair to label *b* as neutral. Considering the "criticality" of the smell, the transformations are more likely to be considered negative because the smelly structure is worse than before. Therefore, it is very important to know if neutral batch refactorings changed the types of the code smells in question, and if they changed, we have to know if it is for worse or for better structure.

Table 5.5: Frequency of neutral batch refactorings affecting smells

| **Heuristic** | **Same Code Smells** | **Different Code Smells** |
|---|---|---|
| Element-based | 11,264 (99.7%) | 30 (0.3%) |
| Range-based | 2020 (100.0%) | 0 (0.0%) |
| Version-based | 6019 (100.0%) | 0 (0.0%) |

In order to mitigate the risk of misclassifying neutral refactorings as illustrated above, we went through our entire dataset to verify the smells present before and the smells present after each neutral batch refactoring. Table 5.5 presents, for each refactoring, how many neutral batches changed the code smells on their scope. As we can see, we observed only 30 changes of code smells in a set containing almost 20,000 batches. These cases are negligible when we are analyzing thousands of batches because they have no power to change our findings regarding neutral batch refactorings. In this way, after this verification, we are confident that neutral batch refactorings are, actually, neutral. These results mean the findings reported in Section 3.3 are also valid in the context of batches. Hence, this leads us to our next finding:

**Finding 6**: Developers tend to produce smell-neutral refactorings when performing either single or batch refactoring.

An analysis of Table 5.4 confirms there is indeed a general trend: independently of the batch synthesis heuristic, neutral refactorings are much more frequent than positive and negative refactorings. When we analyze each individual project, the same classification distribution is observed, i.e., neutral refactorings represent the vast majority in all the projects. These findings suggest developers need more guidance to remove a code smell once they start restructuring a smelly element.

### 5.2.4 Stinky Batches

More than 30% of the batches were found to be negative (or *stinky*) in two synthesis heuristics. These batches induced an increase of smells in their scope. This results is surprising since one might expect that after a sequence of refactorings, a developer would reduce the density of smells. As discussed in Section 3.5, sometimes developers have to apply several refactorings to remove a code smell completely. Thus, this results indicates that even when developers apply multiple refactorings, they still do not remove smells. In this sense, we were wondering if in the future commits, developers succeed in removing these smells. When we analyzed the commits performed after the negative batches, we also concluded that more than 95% of batch-induced smells were not removed afterwards. Only around 14% of batches removed smells, according to two of the heuristics in Table 5.4.

Negative batches were more frequent than positive ones according to two synthesis heuristics. On the other hand, stinky effects are more frequent than positive ones in the context of both scope and version-based synthesis heuristic. These results enable us to answer RQ$_1$: batches made by developers in real projects are not often removing code smells. On the contrary, most of the batches are neutral or negative. This observation also prevails if we only consider refactoring types that, according to their description in Fowler's catalog [3], are explicitly associated with specific code smell types addressed in our study. For instance, the mechanics for applying *Move Method*, *Pull Up Method* and *Move Field* refactorings are associated with smells that represented methods or fields that are misplaced. The misplacement of these members are captured by occurrences of either *Feature Envy*, *Divergent Change*, *Shotgun Surgery* or *God Class*. Hence, the answer to our RQ$_3$ can be summarized by the next finding.

**Finding 7**: Batch refactorings are not often reducing the density of code smells. In fact, most of the batches are neutral or negative.

Our data suggest that most batches are not removing smells. There are possible interpretations of this finding. First, critical design problems in a program may not be related to code smells. If so, this fact may explain why developers either neglect or introduce code smells through refactoring. However, previous studies [5, 9, 34, 50] indicate that design problems are often located in modules containing two or more code smells. Second, similarly to previous studies, we use metrics and thresholds to detect all smells. The proper choice of metrics and thresholds may be sensitive to particular developers [44] and other project-specific factors [74]. As a consequence, our detection of code smells may not reflect what developers truly consider as smells. However, our previous studies involving developers [44, 74] suggest that heuristics used by developers are often not essentially different from smell detection strategies adopted in our study. Our understanding is that refactoring indeed ignores or introduces technical debt in the source code. Finally, it may be the case that our set of studied refactoring types are not among those used by developers to actually remove design problems. Still, it is troublesome that developers introduce smells through batches regardless its synthesis heuristic.

Unfortunately, we do not know why batch refactoring are not removing code smells. Maybe an explanation for this incapability of removing smells is that these batches are somehow incomplete, *i.e.*, developers are not completing the batches with other refactorings to remove the smells completely. As we discussed in Section 3.5, there are some cases that developers need to apply several refactorings to remove a smell. Thus, they are not succeeding in applying these several refactorings to fully remove a smell, even when they apply batch refactoring; or even worse, developers are somehow introducing smells during these batches. Therefore, we need to conduct a further investigation about these cases where batch refactoring introduced and removed smells. This investigation can help us to understand when and why developers do not succeed in removing smells when applying batch refactoring. We can use this knowledge to provide better support when developers apply multiple refactorings. We explain the results of this further investigation in the next section.

## 5.3 Batch-Smell Patterns

To address our next research question, we analyzed the patterns emerging from the relationship between batches and smells. For this purpose, we defined in Section 4.4 two categories of patterns: removal and creational patterns. We consider two patterns instead of three pattern as we did when we considered single refactoring. We focus on removal and creational patterns since they are the ones that can provide recommendations for when developers apply batches. For instance, the knowledge about creational patterns make developers informed about the risks of introducing certain smells along batch refactoring.

In this section, we present the batch-smell patterns according to the smell types. In this way, we can discuss the particularities of each smell related to both creational and removal patterns. We focus on discussing patterns in which more than 50% of the instances of a batch type was related to instances of a specific smell type. For these patterns, we inspected several pattern instances to understand what happened in each case. In particular, we also confirmed whether the batches were directly related to the removal or introduction of the smell. In the following sections we present the patterns related to the code smells *Feature Envy*, *God Class*, and *Complex Class* only. The Appendix A presents the patterns found in the context of other code smells.

### 5.3.1 Feature Envy

Feature Envy is a code smell that happens when a method seems more interested in a class other than the one it actually is declared [3]. It is not rare to come across with a method that invokes several getting methods on another object to calculate some value, representing a *Feature Envy*. Actually, the *Feature Envy* smell was the one observed more frequently in our dataset. Therefore, this smell is involved in several patterns.

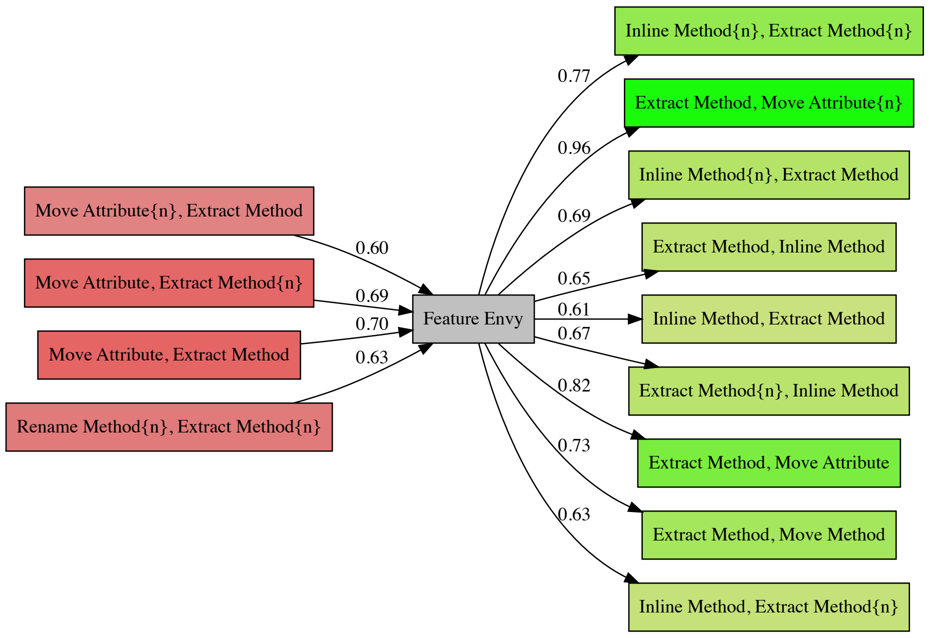

Inline Method{n}, Extract Method{n}
Extract Method, Move Attribute{n}
Inline Method{n}, Extract Method
Extract Method, Inline Method
Inline Method, Extract Method
Extract Method{n}, Inline Method
Extract Method, Move Attribute
Extract Method, Move Method
Inline Method, Extract Method{n}
Move Attribute{n}, Extract Method
Move Attribute, Extract Method{n}
Move Attribute, Extract Method
Rename Method{n}, Extract Method{n}
Feature Envy
0.60
0.69
0.70
0.63
0.77
0.96
0.69
0.65
0.61
0.67
0.82
0.73
0.63

Figure 5.3: Feature Envy patterns

Figure 5.3 presents all batch types involved in some pattern related to the Feature Envy code smell. The red boxes (those to the left side of the *Feature Envy* box) represent the creational patterns, while the green ones (those to the right side of the *Feature Envy* box) represent the removal patterns. The boxes' content represents the batch type involved in the pattern, but there is a caveat regarding the repetition structure. The $\{n\}$ symbol indicates that the refactoring type it succeeds was observed more than one time. The edge weights indicate the frequency that we observed the patterns. For instance, let us consider the top-left red box. The relation of this pattern with the *Feature Envy* code smell can be interpreted as follows: in 60% of the times we observed a batch composed by more than one *Move Attribute* followed by an *Extract Method*, a new *Feature Envy* was created. The same rationale can be used to interpret the removal patterns. The top-right green box indicates that in 77% of the times we observed a batch composed by more than one *Inline Method* followed by more than one *Extract Method*, one instance of *Feature Envy* was removed.

We noticed cases of batches consistently introducing instances of *Feature Envy* in 31 different projects. Figure 5.3 shows that types of batches comprising *Move Attribute, Extract Method* introduce *Feature Envy* smells in more than 60% of the cases. These creational patterns indicate that the batches are somehow incomplete, which contributed to the introduction of the *Feature Envy*. For instance, in the three first creational patterns, the developers moved attributes; however they did not moved the corresponding extracted methods. Consequently the "unmoved methods" become more interested in the classes to which the attributes were moved, leading to the introduction of a *Feature Envy*. In other words, these batches led to the introduction of the *Feature Envy* because they are incomplete; a *Move Method* should also be part of these batches.

Even when considering batches, *Extract Methods* play a central role in the introduction of the referred type of code smell. The behavior should be highlighted is that developers consistently introduce *Feature Envy* code smells while performing *Extract Method* refactorings if they do not complete the batch with a *Move Method*. This is consistent with the single-refactoring patterns presented in Section 3.4.2, where we reported developers introducing *Feature Envy* code smells in 63% of the times while applying *Extract Method*. Our hypothesis in this single-refactoring scenario was that developers introduced *Feature Envy*, but it could be the case those refactorings were only the beginning of the work, and future changes could remove those freshly-introduced *Feature Envy*. After analyzing the restructuring work as a whole (batch refactoring), we still found cases where *Extract Methods* were involved in the introduction of *Feature Envy*.

This result about *Extract Methods* shows that we need to analyze batches in order to understand the impact of refactoring on smells. For instance, through this analysis, we know that *Feature Envy* was introduced because data concerning a method were moved to other classes, but the corresponding method was not moved, as mentioned. Consequently, we can theorize that these negative batches are frequently associated to mistakes or negligence of developers

While analyzing only single refactorings, we found just a few cases of patterns removing *Feature Envies* (Section 3.4.1). According to Fowler [3], *Extract Methods* also play a central role in the removal of *Feature Envies*. While studying single refactorings, we found an alarming low rate of *Feature Envy* removal after performing *Extract Methods* (11%). However, the results presented in Figure 5.3 show nine batch-smell patterns inducing the *Feature Envy* removal. These patterns show us how developers apply smell-removing refactorings in real scenarios, giving us the opportunity to learn how *Feature Envies* are removed during software development.

The results lead to two interesting observations. First, *Extract Method* refactorings play a central role either in the introduction and removal of *Feature Envy*. In fact, when developers apply *Move Method* and *Extract Method*, they tend to neglect *Move Method*, which leads to the introduction of *Feature Envy*. Second, while considering batches, many removal-patterns emerge as opposed to considering only single refactorings. Many could expect that a single refacoting (*e.g.*, *Move Method*) would often symply used to remove a *Feature Envy*.

The outstanding characteristic of these patterns is that the *Extract Method* refactoring is present in all batches. In this way, we can summarize these findings as follow.

**Finding 8**: *Extract Methods* play a central role in the introduction and removal of *Feature Envies*. Several removal patterns of Feature Envies could be only be observed through the analysis of batch refactorings.

As mentioned before, the new patterns we found led us to learn about how refactorings are applied in order to remove specific types of code smells. In fact, we can extract heuristics from these patterns that can guide developers during the removal of code smells. For instance, we can use a combination of *Extract Method* and *Move Method* to remove a *Feature Envy*. Also, we can apply several *Move Attributes* and several *Move Methods* to accomplish the same task, if needed. These data can be used to help developers during smell-removing tasks. In fact, we present a sketch of such heuristics and some preliminary results in Chapter 6.

### 5.3.2 God Class

Our second scenario of batch-smell patterns concerns the *God Class* code smell. As presented in Section 2.2, a *God Class* exists when a class accumulates several responsibilities [3]. We found out that this smell is more frequent than one might expect. We found in our dataset, 425 distinct instances of *God Class* distributed into 26 out of 48 projects. By analyzing these instances, we were able to identify some patterns related to the *God Class* code smell even when analyzing only single-refactorings (Section 3.4.1). In this Section, we present all patterns found during the batch refactoring study, as presented in Figure 5.4.

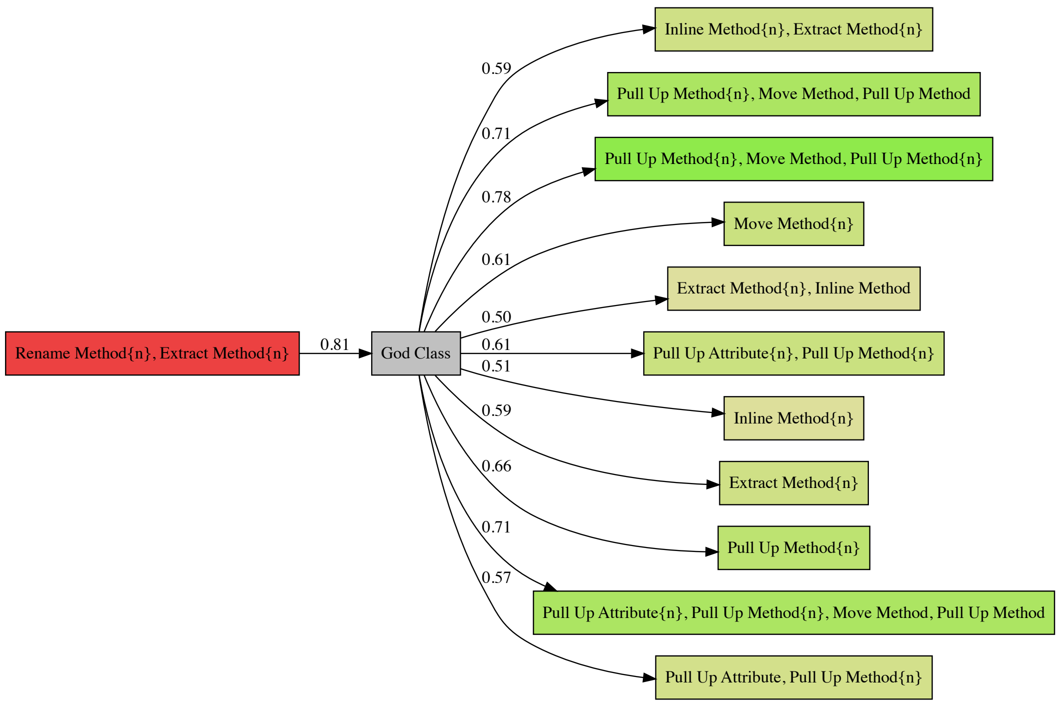

Rename Method{n}, Extract Method{n}
0.81
God Class
0.59
Inline Method{n}, Extract Method{n}
0.71
Pull Up Method{n}, Move Method, Pull Up Method
0.78
Pull Up Method{n}, Move Method, Pull Up Method{n}
0.61
Move Method{n}
0.50
Extract Method{n}, Inline Method
0.61
Pull Up Attribute{n}, Pull Up Method{n}
0.51
Inline Method{n}
0.59
Extract Method{n}
0.66
Pull Up Method{n}
0.71
Pull Up Attribute{n}, Pull Up Method{n}, Move Method, Pull Up Method
0.57
Pull Up Attribute, Pull Up Method{n}

Figure 5.4: God Class patterns

As surprisingly observed in Figure 5.4, batches containing *Rename Methods* and *Extract Methods* were responsible for creating *God Classes* frequently. At the first sight, the relation between these refactoring types and the creation of this kind of code smell is not intuitive, since developers are not expected to increase the size of classes while performing *Rename Method* and *Extract Methods*. In order to understand why this batch led to the introduction of *God Class*, we analyze these batch instances. This creational pattern exists because it often happened in the context of floss refactoring; thus, developers interleaved additional changes while refactoring. During these additional changes, they introduced the *God Class*.

This result indicates that when developers apply several *Extract Methods*, the risk of creating *God Classes* exists as they might be too focused on their non-structural additional objectives. What is happening in these cases is the increased number of methods are making explicit that may many methods were grouping unrelated functionalities. As a consequence, the number of responsibilities increases through the growth of new methods, and the class cohesion decreases. Consequently, it leads to the appearance of *God Class*. Indeed, this smell exists because a class implements too many functionalities, and it is interesting to notice how this characteristic is reflected in the creational patterns. For instance, in our single refactoring study, we found three creational patterns related to this smell: *Move Method*, *Pull Up Field* and *Pull Up Method* (Section 3.4.2). On the other hand, in the batch refactoring, we found one pattern: *Rename Method* with *Extract Method*. Notice that there is not intersection of refactoring types between single and batch refactoring patterns; however, all these refactorings are related to moving data between different entities. This indicates that a *God Class* can be introduced when developers perform these transformations where attributes and methods are moved between elements, specially when it happens in the context of floss refactoring.

We can notice that this behavior of moving data is also reflected in the removal patterns. We identified 11 removal patterns related to *God Class*, and all them use refactorings that move data between code elements. With the exception of *Inline Method* and *Extract Method*, all the other refactorings are related to moving data between different entities. This results confirms what we discussed above about the creation of *God Class*. This smell can be created in the context of floss refactoring as mentioned; thus, any additional changes can lead to its introduction. Conversely, to remove this smell, developers will need to apply different refactoring types to move the data to the elements that suit them better. Consequently, we will have several removal patterns, as confirmed by our results.

In addition to identify both patterns that remove and introduce code smells in practice, we highlight that these patterns have not been reported elsewhere. Even Fowler's catalog [3], which list common refactorings to remove *God Class*, does not report these patterns. Fowlers' catalog indicates that developers should apply *Extract Class* or *Extract Subclass*. However, we noticed that in practice, developers much more often follow other strategies by applying other refactoring types: *Inline Method*, *Extract Method*, *Pull Up Method* and

*Attribute*, and *Move Method*. This results is interesting since it shows what refactoring catalogs [3] say may not be what happens in practice. The data reported in Figure 5.4 lead us to our next finding as follows.

**Finding 9**: Refactorings that move data play a central role in the addition and removal of *God Classes*. This is even more evident in the context of floss refactoring, where developers interleave the refactorings with additional changes. These changes may deviate the attention of the developer from noticing they are inducing the emergence of a *God Class*. This behavior forces developers to apply different types of refactorings to move the data to appropriate elements, consequently creating several removal patterns performed later.

### 5.3.3
### Complex Class

A *Complex Class* is a code smell that indicates when a class starts to become harder to understand and maintain. In this work, we consider a *Complex Class* all of the ones that have at least one method with high cyclomatic complexity [36]. Interestingly, this was one of the most common code smells present in our dataset. In fact, this was the most common one if we consider only smells related to the class-level, not method-level smells, such as *Long Method* and *Feature Envy*. This code smell affected 43 out of our considered 48 projects. Since this smell is so common, and affect so many projects, we could also observe several batch-smells patterns.

Figure 5.5 presents all batch-smells patterns found that are related to *Complex Classes*. As we can observe, the study revealed fifteen removal, and only one creational patterns. As one could expect, the method-moving refactorings play a central role when we consider creation and removal of *Complex Class*. Interesting enough, a reason for the class to have a high cyclomatic complexity is the implementation of several functionalities. Thus, it is reasonable that the creational pattern that introduces *Complex Class* is the same that introduce *God Class*. Consequently, the same discussion about moving data between code elements in the context of *God Class* also applies in the context of *Complex Class*. Indeed, all *Complex Class* patterns contain either method-moving refactorings or *Extract Methods*, independently on being removal or creational.

As described before, a class is considered to be complex when at least one method has high cyclomatic complexity. So, in our study, if the developer reduces the cyclomatic complexity (CC) of the methods in a particular class,

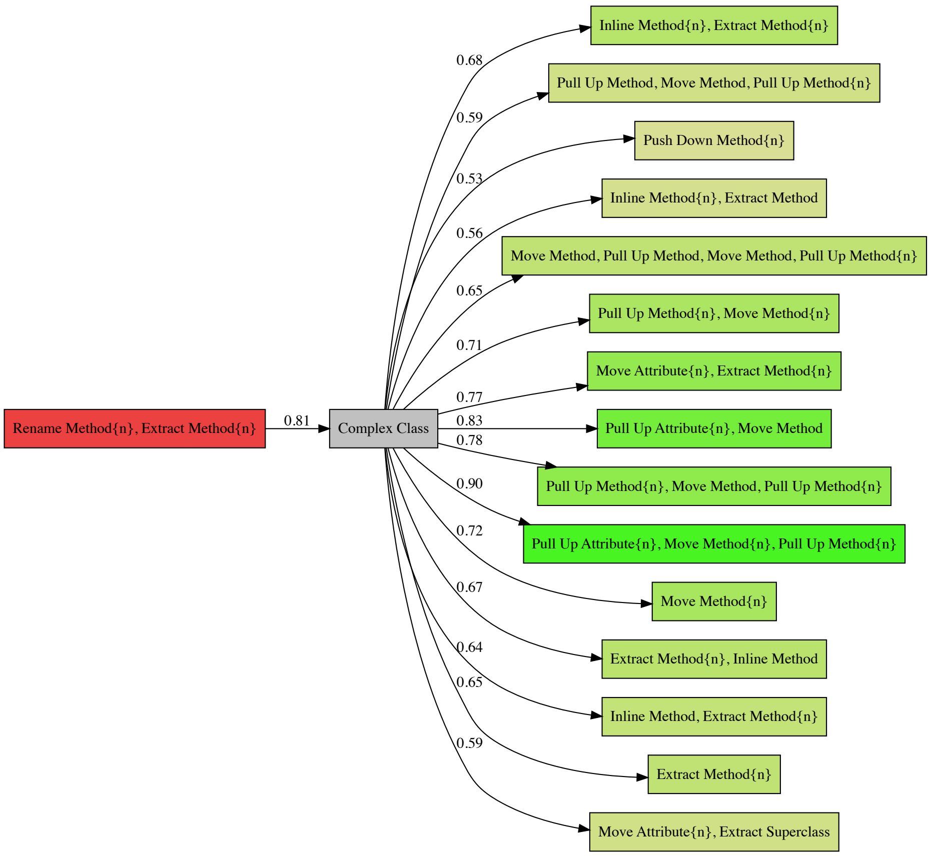

Rename Method{n}, Extract Method{n}
0.81
Complex Class
0.68
Inline Method{n}, Extract Method{n}
0.59
Pull Up Method, Move Method, Pull Up Method{n}
0.53
Push Down Method{n}
0.56
Inline Method{n}, Extract Method
0.65
Move Method, Pull Up Method, Move Method, Pull Up Method{n}
0.71
Pull Up Method{n}, Move Method{n}
0.77
Move Attribute{n}, Extract Method{n}
0.83
Pull Up Attribute{n}, Move Method
0.78
Pull Up Method{n}, Move Method, Pull Up Method{n}
0.90
Pull Up Attribute{n}, Move Method{n}, Pull Up Method{n}
0.72
Move Method{n}
0.67
Extract Method{n}, Inline Method
0.64
Inline Method, Extract Method{n}
0.65
Extract Method{n}
0.59
Move Attribute{n}, Extract Superclass

Figure 5.5: Complex Class patterns

the code smell would be eventually removed. For instance, let us assume a particular class containing a method with CC = 20 in a case where CC >= 10 indicates a complex class. In this scenario, the developer would need to break this method into three or more methods where CC < 10 to get rid of the complex class. The refactoring type that suits better, in this case, is the *Extract Method*, so this explains most of the removal patterns containing this refactoring type. When the developers face a highly complex method, they simply break it into smaller and less complex ones.

A curious fact in these patterns is the massive presence of method-moving refactorings, such as *Move Method*, and *Pull Up Method*. Those refactorings per se do not reduce the complexity of classes since they only move methods from one class to another. However, it is worth mentioning most of the refactorings we observed use the floss refactoring tactic (Section 2.1.5). So, in addition to moving the methods to different classes, the developers also apply different changes to reduce the complexity. If they apply method-moving refactorings by using the root-canal tactic, they would only transfer the code smell from one class to another. Therefore, the data reported in Figure 5.5 lead us to our next finding as follow.

**Finding 10**: *Extract Methods* act as a complexity reducer and help developers to get rid of *Complex Classes*. Also, moving-method refactorings that use the floss-refactoring tactic contribute to the removal of such complex classes.

## 5.4 Threats to Validity

The batch synthesis heuristics represent a threat to the internal validity because they might lead us to false positives, i.e., batches that, in fact, are not batches. If we chose to use one heuristic that generates several false positives, then our results would not be trustworthy. In order to mitigate such a threat, we used three different heuristics derived from previous studies (Section 4.2). In this way, we can compare the results obtained by the three heuristics. Fortunately, all of them points to the same direction, indicating that our results are reliable. Another way of validating such heuristics is to explore the data they provided to solve a different problem. If a real refactoring problem can be solved by them, then we can interpret this as an additional evidence that the synthesized batches are valid. In fact, we conduct an additional study reported in the next chapter.

Some findings of this study are centered around the high frequency of neutral batch refactorings. However, if our classification scheme is somewhat inaccurate in identifying neutral refactorings, then we have a major threat to the validity of our data. In order to mitigate that, we studied all the cases where the classification scheme could be inaccurate (Section 5.2.3). We found a risk of the classification scheme being wrong on 0.01% of the cases. In this way, this risk was mitigated by the data disposition, i.e., the way developers performed the neutral batches. In this way, in any case of replication of this study, the researcher must execute the same verification and check the validity of the classification.

In this chapter, we also presented several patterns where batches removed or introduced code smells. Precisely, we presented removal and creational patterns related to *Feature Envy*, *God Class*, and *Complex Class*. We computed them by verifying how often they happen in the analyzed projects, so they might suffer from lack of generality. If the patterns are not generic enough, then they cannot be used in different systems. In order to mitigate such threat, we conducted a different study where the patterns are used in a completely different set of projects, giving us confidence about their validity (Chapter 6).

## 5.5 Related Work

Bavota *et al.* [17] mined the evolution history of 3 Java open source projects to investigate if refactorings occur on code elements that certain indicators suggest a need for refactoring. According to their results, quality metrics do not show a clear relationship with refactoring and 42% of the refactorings are applied on smelly elements, in which only 7% of them remove smells. In both studies we executed (single and batch refactorings), the results differ. First, the percentage of refactorings applied to smelly elements are higher in our dataset. Second, batch refactorings and single refactorings remove code smells in more than 7% of the cases independent on the synthesis heuristics. Our results show that developers tend to not remove code smells even when applying a sequence of refactorings (batch).

As mentioned in Chapter 3, Silva *et al.* [15] mentioned that future studies on refactoring recommendation systems should refocus from code-smell-oriented to the maintenance-task-oriented solutions. We still think that such refocus needs more reflection. Instead, a refactoring recommendation system should employ a hybrid solution, in which both code-smell-oriented and maintenance-task-oriented solutions are combined. Thus, an envisaged recommendation system can support developers to conclude their maintenance tasks in a way that smells are also removed or not introduced. This chapter presented several removal patterns, different from what we reported in Chapter 3. This is an advance regarding the knowledge needed to remove code smells from software systems. In fact, we explore such patterns in the next chapter, giving evidence that code smells can be used as an important factor for influencing recommender systems.

## 5.6 Summary

In this chapter, we present a work where we studied the batch refactoring phenomenon. Differently from what was presented in Chapter 3, we here consider groups of refactorings to evaluate their impact on the existence of code smells. Interestingly, the occurrence of a negative impact on code smells is reduced when we study refactorings as groups (batches). However, we still find a high number of negative and neutral refactorings, even considering them as a group. These results were consistent across several projects, and also across all synthesis heuristics we used.

We also present a study regarding batch-smell patterns. We were able to identify several patterns, as presented in Section 5.3. We revealed several removal patterns, that can be used as a knowledge base about how developers remove code smells by using batch refactoring. In fact, the next chapter explores the removal patterns to give one step closer to possible automatic heuristics for smell-removal refactorings.

# 6 Improving Batch Refactoring: Recommendation Heuristics

Developers may introduce all sorts of mistakes when refactoring. They can even unnotice that the structural changes are not sufficient to fully remove smelly structures. Indeed, as discussed in the previous chapter, often batch refactorings are not reducing the number of smells. That happens because batch refactoring is far from being a trivial process. Since developers are not removing code smells when applying batch refactoring, they need support in this process.

We conducted in the previous chapters the first step towards providing such support for developers during refactorings. In these chapters, we presented studies that extracted plenty of data regarding the refactoring practice. The previously reported results help us to understand how developers usually make mistakes and degrade the structural quality of the system via refactoring. Consequently, we can use this knowledge to support developers in at least two ways. First, we can help developers by providing warnings when developers perform refactoring operations. For instance, we can make them aware about refactorings that frequently introduce smells. For this purpose, we can use the patterns presented in Sections 3.4 and 5.3.

A second way to provide support for developers is by helping them to apply a batch refactoring that remove code smells completely. This second way is the focus of this chapter, which we present and evaluate a suite of new recommendation heuristics to help developers to apply batch refactoring. These heuristics are based on the knowledge extracted from the three code smells (*Feature Envy*, *God Class*, and *Complex Class*) related to creational and removal patterns discussed in Section 5.3. Based on these patterns, we proposed three heuristics, each one for of the three aforementioned code smells. After presenting the heuristics, we need to assess their potential usefulness in real scenarios. Therefore, we also performed a first evaluation of the proposed heuristics.

We structured this chapter as follow. First, we present the derived heuristics in Section 6.1. We first present the heuristic used to remove *Feature Envy*, followed by the ones responsible for removing *God Class*, and *Complex Class*, respectively. After this, we present the experiment setup and results in Section 6.2. We present the threats to validity in Section 6.3, while the summary and concluding remarks of this chapter is in Section 6.4.

## 6.1 Smell Removal Heuristics

In this section we present three heuristics for removing code smells. Each heuristic is focused in removing a particular type of code smell. As mentioned before, those heuristics were derived from the batch-smell patterns presented in Section 5.3. As we can observe in the figures presented in that section, we have plenty of possible ways of proposing heuristics. For instance, we can remove a *Complex Class* by applying several *Push Down Methods* or by applying a sequence of *Extract Methods*. Since our objective in this study is to check the viability of deriving useful heuristics, we selected and evaluated only one pattern for each code smell.

### 6.1.1 Removing Feature Envy

One of the most common patterns found for removing *Feature Envy* is composed of *Extract Method* and *Move Method*. Therefore, many times when developers were successful in removing *Feature Envies*, they first extracted the foreign part of the method into a new one and then moved the newly created method to a different class. This procedure roughly describes the heuristic we present for removing *Feature Envies*. Formally, the *Feature Envy* removal heuristic is composed of four parts: (i) identification of method lines that are more interested in different class; (ii) extraction of these lines into a new method; (iii) identification of the class that suits better the newly-created method; and (iv) application of a *Move Method* refactoring to move the new method to the identified class.

Each step of this heuristic poses a different challenge. The first one is that we need to identify lines of the *Feature Envy* method that are more interested in a different class other than the one they are. For several reasons, it is not trivial to recommend lines of code to be extracted from a method. The piece to be removed must execute a particular functionality in a way that must make sense to remove the lines together. Several studies propose different techniques to accomplish the objective of discovering extract method opportunities [57, 59]. The technique proposed by Charalampidou [80] is based on the functional relevance of the combined lines. Their paper introduces an approach that aims at identifying source code chunks that collaborate to provide specific functionality and propose their extraction as separate methods. Since their approach fits very well with our first part for recommending *Feature Envy* removal, we adopt it.

Therefore, to accomplish the first part of this heuristic, we implemented the approach proposed by Charalampidou [80]. As described in their paper, they propose an approach called SRP-based Extract Method Identification (SEMI). In particular, their approach recognizes fragments of code that collaborate for providing functionality by calculating the cohesion between pairs of statements. The extraction of such code fragments can reduce the size of the initial method, and subsequently increase the cohesion of the resulting methods. In our scope, we implemented their technique and treated this component as a black box, where the input is a method and the output is a set of line intervals that can be extracted. For instance, let us consider a method with 5 lines. The algorithm would return possible line intervals that could be extracted, for instance [1,2], [2,4], or [2,5].

By having these possible intervals, we have several possibilities to recommend extraction. However, only having these intervals are not enough to remove the *Feature Envy* since we do not want to recommend an extraction that would still maintain the smell that we already had. In this way, we run a verification step for each interval. We simulate the removal of such lines by disregarding their influence on the *Feature Envy* detection. In this way, we test, for each interval, if its removal would lead to a *Feature Envy*. If the removal of a single interval is not enough to remove the *Feature Envy*, then we look for a combination of two intervals. We keep increasing the number of intervals on the tests until we have a *Feature Envy* removal possibility. After completing this step, we can move on for the second part of the heuristic: *Extract Method* refactoring.

After recommending the extraction, the developer can apply the *Extract Method* refactoring. After this step, we can test if there is still a *Feature Envy*. Unfortunately, there is a risk on only performing this extraction. The newly-created method could have the *Feature Envy*. However, we have to leverage in the fact that we know its lines are cohesive and can be moved together to a different class. In this way, we check what is the class this new method relates the most, either by method calls or attributes use. After discovering this class, we can recommend a *Move Method* refactoring of the newly-created method to this discovered class.

Therefore, our first heuristic is completed by executing the four described steps. We first recommend an *Extract Method* by combining our code smell detection tools with the SEMI approach. After extracting the method, we can recommend a *Move Method* by examining the method calls and attributes use of the newly-created class. In this way, we reproduce programmatically the batch-smell pattern *Extract Method, Move Method* presented in Figure 5.3. Notice that we not only reproduced the removal pattern, instead we made some tests to make sure that the batch would be able of removing the smells. The combination of these tests with the knowledge about the removal patterns is what increases the chance of our heuristic to help a developer to get rid of a *Feature Envy*.

## 6.1.2 Removing God Class

As presented in Section 5.3.2, moving-method refactorings play a central role on *God Class* removals. Therefore, we based on this fact to propose a heuristic to remove this smell. *God Class* is a class that assumes several responsibilities in a system. In Section 5.3.2, we discussed that if we distribute these responsibilities (*i.e.*, methods) over several classes in the system, the developer can remove the smell. Indeed, to perform this distribution, we found that developers can apply different types of refactorings (Section 5.3.2); for instance, he can apply a batch composed of *Move Method{n}*. Hence, the heuristic we implemented to remove God Class is based simply on method-moving refactorings.

According to the rules of *God Class* detection [17], a class is considered affected by this smell if it has cohesion lower than the average of the system and more than 500 lines of code. This threshold can be tailored to particular projects or modules by using machine learning techniques, as we did in a recent work [44]. In this way, for each method in the class, we identify a suitable class to which we move it. We used the same strategy presented in Section 6.1.1 to identify the class of destination of the method. We keep recommending *Move Methods* until the *God Class* is removed. However, such operations pose a risk of creating new *God Classes* in the system, as presented in Table 3.9. Therefore, before recommending a Move *Method*, we check if the destination class would become a *God Class*. If this is true, we simply change the recommendation to the second most suitable class found.

It is worth mentioning that we find the suitable class by counting the number of calls and accesses to attributes. For instance, let us assume that a particular method $m$ calls 3 methods and accesses 2 attributes from class $A$. In this case, the “bonding factor” of $m$ to $A$ is 5. Let us assume the same method $m$ calls 4 methods from the class $B$, leading to a bonding factor of 4. Now, assume our heuristic recommended to move $m$ to $A$, but $A$ would be transformed into a *God Class* if this occurs. In this case, the heuristic would recommend the method-moving change to class B, since in this case, $B$ would be still smell-free.

In summary, the second heuristic is a sequence of *Move Methods*. However, it also uses the smell-detection tool to understand when the target class would not be a *God Class* anymore. Additionally, the smell detection tool is used to prevent the creation of new code smells after the recommended refactoring.

### 6.1.3 Removing Complex Class

In our studies, we consider a class as *Complex Class* if it has at least one method having a high cyclomatic complexity (CC) [36]. So, the strategy to remove such smell is related to the reduction of the complexity of methods with high CC. As presented in Section 5.3.3, developers often apply *Extract Methods* to remove such complex structures. Hence, the heuristic to remove *Complex Class* is composed of four parts: (i) identify all methods with high CC; (ii) identify *Extract Method* opportunities to reduce the complexity; (iii) evaluate the identified opportunities; and (iv) recommend *Extract Methods*.

The first part is implemented by our code smell detection tool. We made a simple modification in order to find *Complex Methods* in a particular *Complex Class*. After finding them, we use the SEMI approach presented in Section 6.1.1 to generate possible line intervals to be extracted. After identifying such intervals, we need to evaluate the identified opportunities. For each interval found, we simulate its removal and compute what would be the new complexity of the method. When we find a interval (or a set of interevals) that reduces the complexity, we start recommending the *Extract Methods*.

Therefore, after running the steps of this heuristic, our tool can identify pieces of code that can be extracted to reduce the complexity of the methods found. After recommending a batch of *Extract Methods*, we can distribute the complexity of the class into several smaller methods, getting rid of the original *Complex Class*. It is worth mentioning the recommended extractions can pose a risk and create new code smells, such as a new *Feature Envy* (Section 5.3.1). If this occurs, we can trigger the *Feature Envy* removal heuristic to improve the batch by removing the introduced smell.

## 6.2 Heuristics Evaluation

Section 6.1 introduces three heuristics we derived from the patterns presented in Section 5.3. As a way of checking if these heuristics have the potential to be used in practice, we designed and executed a quasi-experiment [81]. This section presents the experimental tasks along with the key results.

### 6.2.1 Goal and Research Question

The heuristics presented in Section 6.1 aim at removing three different types of code smell: *Feature Envy*, *God Class*, and *Complex Class*. We hypothesize these heuristics can be useful in real scenarios, but we have no evidence so far to neither confirm nor refute this hypothesis. Therefore, we designed a quasi-experiment to evaluate the usefulness of these preliminary heuristics. In this context, the goal of this last study is:

**Goal:** Evaluate the smell-removing heuristics derived from the batch-smell patterns.

In order to evaluate the usefulness of the heuristics, we have to observe how they perform in real scenarios. In addition to this, we must apply them to new projects, i.e., projects not analyzed by our previous studies. Since these heuristics were derived from the projects presented in Tables 3.1 and 3.5, if we applied them back to the same projects, we would have a reasonable chance of getting good results. Hence, to mitigate such risk, we evaluate the usefulness of the smell-removing heuristics in different projects. In this context, this study aims at addressing the following research question:

Are the smell-removing heuristics able to improve the code structural quality?

To address this research question, we executed a quasi-experiment. We first apply the heuristics steps on different smelly code elements. As presented in Section 6.1, each heuristic comprises some steps, and the application of each step in a code element delivers a new code state. In this way, we documented each code state obtained as a result of the application of the heuristics steps. After this, we compiled all the results and asked for the opinion of 20 software developers. They had to evaluate the code states and inform us of their opinion about the impact of the code changes on the code structural quality. After this, we analyzed the results in order to answer this research question.

### 6.2.2 Experimental Tasks

To evaluate the heuristics, we defined four activities, as described next.

**Activity 1: Sample Selection.** The first part of the experiment was the sample selection. Since the objective is to evaluate the heuristics, we had to execute them in different classes affected by the studied code smells, then we selected 3 different classes for each smell. In this way, we executed each of the proposed heuristics in the contexts of three classes containing the corresponding smells. We selected three classes for each smell because the experiment's participants could have proper time to inspect each of the 9 recommended batches. Also, we selected, for each smell, classes from three distinct projects in order to ensure some heterogeneity to the sample. Besides that, we chose classes implemented for different purposes, from log-in services to classes that manage students data from an educational institution.

**Activity 2: Heuristics Execution.** We then executed the heuristic steps described in Section 6.1 for each sample. As presented in that section, each heuristic produces as output a list of recommended refactorings. For instance, let us go through all steps of one of the batches generated for this experiment. The code smell detection tool flagged the *replaceImages* method presented below as *FeatureEnvy*, as it seems to be more interested in the class of the object *images* rather than the class it was declared. In this way, we executed the heuristic steps on it. As described in Section 5.3.1, the first step is to recommend an *Extract Method*. One of the recommendations of the heuristic was to extract from the method body of *replacedImages*. The extracted code is marked in light gray in the code below:

```java
public class Media {
  ...
  public void replaceImages(PicturePool oldPool) {
    if (this.images == null) {
      return;
    }
    images.setId(oldPool.getId());
    images.getLowRes().setId(oldPool.getLowRes().getId());
    images.getStdRes().setId(oldPool.getStdRes().getId());
    images.getThumb().setId(oldPool.getThumb().getId());
  }
  ...
}
```

The output of the first refactoring can be seen in the code snipped below. According to the heuristic, the method *updatePool* is still a *Feature Envy*. The first refactoring was not considered to be enough to solve the problem. In this way, the heuristic recommends a new *Move Method* refactoring to host the newly-created method in a suitable class.

```java
public class Media {
  ...
  public void replaceImages(PicturePool oldPool) {
    if (this.images == null) {
      return;
    }
    updatePool(oldPool);
  }

  private void updatePool(PicturePool oldPool) {
    images.setId(oldPool.getId());
    images.getLowRes().setId(oldPool.getLowRes().getId());
    images.getStdRes().setId(oldPool.getStdRes().getId());
    images.getThumb().setId(oldPool.getThumb().getId());
  }
  ...
}
```

The heuristic identified that the *updatePool* method is more interested in the class of the object *images* rather then the *Media* class. It turns out this class is named *PicturePool*. So, *updatePool* method is moved to the *PicturePool*, as presented below.

```java
public class Media {
  ...
  public void replaceImages(PicturePool oldPool) {
    if (this.images == null) {
      return;
    }
    images.updatePool(oldPool);
  }
  ...
}

public class PicturePool {
  ...
  public void updatePool(PicturePool oldPool) {
    this.setId(oldPool.getId());
    this.getLowRes().setId(oldPool.getLowRes().getId());
    this.getStdRes().setId(oldPool.getStdRes().getId());
    this.getThumb().setId(oldPool.getThumb().getId());
  }
```

```
    ...
}
```

After these transformations, the heuristic certifies that the *Feature Envy* was removed by running our code smell detector again. Since neither the original *Feature Envy* is present nor new code smells were introduced, then the heuristic stops executing. The example presented in this section is an execution of the *Feature Envy* removal heuristic. However, we have three different heuristics, and each one was executed in three different smelly code elements. All executions produce a sequence of refactorings that, according to our code smell detector, remove the smelly structure. In order to present the execution of the heuristics to the developers, we compiled all code transformations, all code smells, all refactorings applied and presented them in a web page.

**Activity 3: Subjects Characterization.** We asked the developers to fill out a questionnaire to gather their information, including educational level, professional experience with software development in terms of years, experience with Java programming (in years), and whether they are familiar with code smells and refactoring or not. The data collected during this activity was used to understand if the participants meet the minimum requirements needed to participate in the experiment. Since all code examples are in Java, the participants have to be able to read and understand the code. Also, they have to know how to refactor a piece of code. Otherwise, it would be very hard for them to understand the heuristics steps, invalidating their answers. Screen shots of the questionnaire are available in Appendix C.

**Activity 4: Experiment Execution.** As mentioned before, we executed the heuristics steps on 9 different smelly elements. Each execution led us to a sequence of refactorings, implicating in several code changes. Each participant had to evaluate each sequence of refactorings generated for each one of the 9 classes. In this way, each participant had to visualize and evaluate 9 batch refactorings. After visualizing each batch, the participants had to answer the following question:

> What is your opinion about the impact of the sequence of refactorings on the code structure quality?

As we can see, the question we made to the participants does not involve the term *code smells*. Although the heuristics had been derived from relationships between batches and code smells, we are ultimately interested in improving the code structural quality. Since from the first study, we used code smells as an indicator of structural problems. If developers feel that the code had its structural quality improved by our heuristics, this is one more evidence that code smells are, in fact, good estimators to measure the code structural quality. In other words, we can keep developing heuristics focusing in code smells because, in the end, the code structural quality can be improved if we remove them.

Table 6.1: Possible answers during the quasi-experiment

| **Answer** | **Description** |
|---|---|
| Positive | The code structural quality has improved |
| Intermediate | There are benefits, but I think there is room for improvement |
| Negative | The code structural quality has decreased |

The participants had to choose between Positive, Intermediate, and Negative as an answer to the provided question. In each case, they had to justify the answer. Table 6.1 presents the three possible answers that each participant had to give for each one of the nine presented batches. In any case, they had always to provide a justification for the answer in an open text field, so we can use it to better understand their answers. We developed a web application in order to present batches and to collect the developers' answers.

## 6.2.3 Data Presentation and Analysis

In this section, we first present the summary of the answers to the characterization questionnaire (Section 6.2.3.1). Later, in Section 6.2.3.2, we present the results of our quasi-experiment.

### 6.2.3.1 Characterization Questionnaire Data

We invited 20 software developers to answer our quasi-experiment. Section 6.2.3.1 presents the data regarding the participants' years of experience with software development, years of experience with Java, and number of Java developed projects. As we can see, most of the participants have experience in industry both with software development, and with the Java language. Only one participant have no experience in industry. However this participant have experience with code smells and refactoring research. In this way, the lack of experience do not pose a threat to the answers provided by this participant.

As mentioned in Section 6.2.2, we asked for the participants to answer the characterization questionnaire in order to verify if they attend the minimum requirements to participate in our experiment. If a participant has experience

Table 6.2: Participants' characterization data

| Answer | Median | Average | Std. Dev. | Max | Min |
|---|---|---|---|---|---|
| Years of experience with software development | 4 | 5.5 | 4.6 | 15 | 0 |
| Years of experience with Java | 1 | 2.8 | 3.9 | 14 | 0 |
| Number of Java developed projects | 1 | 5.4 | 13.4 | 50 | 0 |

with software development, have, at least, little knowledge of Java, and is familiar with refactoring, then we consider the participant apt. Therefore, all participants in our experiment are capable of answering the questions reasonably well. In addition to that, all participants said to be knowledgeable about refactoring, and also all of them apply refactorings often. We also observed that 85% of them often removes code smells from their source code, even though 100% of then knows about code smells.

#### 6.2.3.2 Quasi-experiment Results

As mentioned before, the developers had to evaluate each one of the 9 batches produced by the smell-removing heuristics. In each evaluation, they had to inform their opinion about the impact of the batch on the code structure quality. Figure 6.1 presents the data regarding the answers given by the developers. This figure is divided into three sets of bars. The first one presents the data regarding *Complex Class*, while the second one presents the answers about *Feature Envy*. Finally, the *God Class* data is presented in the last set. As we can observe, the developers were very positive about the outcomes of the heuristics. Considering the entire dataset, 73% of the answers were *positive*. In this way, we conclude that more than 93.9% of the answers state the recommended refactorings improve the code quality at least partially (*positives* plus *intermediate*). Out of 180 answers, only 11 were *negative*.

Table 6.3: Participants' answers by each class and smell considered

| Class | Code Smell | Negative | Intermediate | Positive |
|---|---|---|---|---|
| Clause | Complex Class | 15.0% | 35.0% | 50.0% |
| DiarioClasseService | Complex Class | 0.0% | 15.0% | 80.0% |
| GenericTranspalBean | Complex Class | 10.0% | 5.0% | 80.0% |
| IngressoUniversidadeService | Feature Envy | 5.0% | 30.0% | 65.0% |
| Media | Feature Envy | 0.0% | 10.0% | 90.0% |
| UserFactory | Feature Envy | 5.0% | 20.0% | 75.0% |
| EmployeeUtils | God Class | 10.0% | 20.0% | 70.0% |
| LibraryMainControl | God Class | 5.0% | 20.0% | 75.0% |
| MatriculaAcademicaService | God Class | 5.0% | 20.0% | 75.0% |

This positive outcome is also consistent if we observe the data for each batch evaluated. Table 6.3 presents the answers of the participants by considering each class individually. In this table, the first column presents the name of the class used to apply the heuristic step. The second column presents the code smell type that affected the corresponding class. The last three columns present the percentage of negative, intermediate, and positive answers, respectively. As we can observe, all batches had very few negative answers. These results give us confidence that we can derive useful heuristics from the patterns presented in Section 5.3. It is worth mentioning the *Clause* class was the one with the most negative and intermediate answers, so we discuss this particular case later in this chapter. From now on, we will discuss the results for each one of the heuristics.

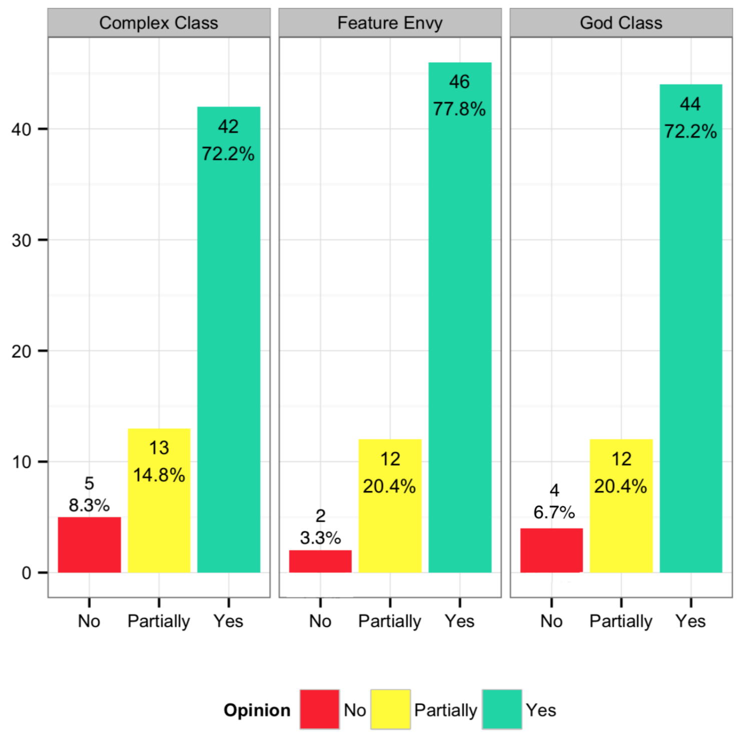


Figure 6.1: Experiment answers

**Complex Class Results.** The heuristic for recommending refactorings of *Complex Class* can be considered the least successful one if we consider its 8.3% against the 6.7% for the *God Class* heuristic. Most of these negative answers were concerning a particularly difficult case. In this example, the developers evaluated the recommended refactorings in the *Clause* class presented below. This class contains several methods and attributes, but we present below only the problematic one, i.e., the *toCriterion()* method.

```
public class Clause {
  ...
  public Criterion toCriterion() {
    if (operator.equals(Operator.IS_NULL)) {
      return Restrictions.isNull(queryingField);
    } else if (operator.equals(Operator.EQUALS)) {
      return Restrictions.eq(queryingField, this.value);
    } else if (operator.equals(Operator.IS_NOT_NULL)) {
      return Restrictions.isNotNull(queryingField);
    } else if (operator.equals(Operator.NOT_EQUALS)) {
      return Restrictions.ne(queryingField, this.value);
    } else if (operator.equals(Operator.GREATER)) {
      return Restrictions.gt(queryingField, this.value);
    } else if (operator.equals(Operator.GREATER_EQUAL)) {
      return Restrictions.ge(queryingField, this.value);
    } else if (operator.equals(Operator.LESSER)) {
      return Restrictions.lt(queryingField, this.value);
    } else if (operator.equals(Operator.LESSER_EQUAL)) {
      return Restrictions.le(queryingField, this.value);
    }
    throw new FilteringException();
  }
  ...
}
```

The *Clause* class was flagged as *Complex Class* because of the method *toCriterion*, which has a high cyclomatic complexity. As we can see, this method is composed of a chain of ifs, leading to a low code readability. As presented in Section 6.1.3, the heuristic recommends a list of extract methods to distribute the complexity. In this particular case, the heuristic recommended two extract methods, and the final result can be seen below.

```
public class Clause {
  ...
  private Criterion getEqualityCriterion() {
    if (operator.equals(Operator.IS_NULL)) {
      return Restrictions.isNull(queryingField);
    } else if (operator.equals(Operator.EQUALS)) {
      return Restrictions.eq(queryingField, this.value);
    } else if (operator.equals(Operator.IS_NOT_NULL)) {
      return Restrictions.isNotNull(queryingField);
    } else if (operator.equals(Operator.NOT_EQUALS)) {
      return Restrictions.ne(queryingField, this.value);
    }
    return null;
  }

  private Criterion getComparisonCriterion() {
    if (operator.equals(Operator.GREATER)) {
      return Restrictions.gt(queryingField, this.value);
    } else if (operator.equals(Operator.GREATER_EQUAL)) {
      return Restrictions.ge(queryingField, this.value);
    } else if (operator.equals(Operator.LESSER)) {
      return Restrictions.lt(queryingField, this.value);
    } else if (operator.equals(Operator.LESSER_EQUAL)) {
      return Restrictions.le(queryingField, this.value);
    }
    return null;
  }

  public Criterion toCriterion() {
    Criterion equality = getEqualityCriterion();
    if (equality != null) {
      return equality;
    }

    Criterion comparison = getComparisonCriterion();
    if (comparison != null) {
      return comparison;
    }
    throw new FilteringException();
  }
  ...
}
```

The *toCriterion* method was target of two extract methods, leading to the creation of *getComparisonCriterion* and *getEqualityCriterion*. The *Complex Class* was removed because the class no longer has methods with high cyclomatic complexity. However, the legibility of the created methods is still not good, as considered by some developers. They suggested a complete rewrite of this method by using different data structures and even a different object model design to implement the same functionality. Unfortunately, the heuristics are not able to recommend that, since the suggested changes are not a combination of the refactorings presented in Section 2.1.3. Interestingly, the developers who gave *negative* answers to this case were the less experienced ones. The most experienced developers answered as *intermediate*, indicating they are less rigid when looking for structural improvement. Clearly, a total reshape is the best solution for the case, as suggested by the less experienced developers. In the perspective of the most experienced ones, the heuristic could achieve some improvement, but there is still room for making the class structurally better. Now, we quote the most negative answer about this outcome:

> "Honestly, these refactoring operations did not help at all. I do agree the method is less complex, but the complexity was merely spread into the other two methods, i.e., the other methods still have several conditional expressions." – *Developer with 1 year of experience.*

However, even in this hard case, we achieved 10 *positive*, and 7 *intermediate* answers. Some developers agree the complexity was removed and the smell obliterated. Also, several developers said they agree the *Complex Class* was removed, but more refactorings could be applied to improve the method even further. For instance, one of the developers said:

> "Yes, both extract methods helped the method to be more readable and maintainable." – *Developer with 2 years of experience.*

In summary, the vast majority of the answers were positive. One of the batches achieved a rate of 90% of positive answers. Even when the smell is hard to be removed through a sequence of refactorings, the heuristic obtained positive feedbacks, as the *Clause* class presented.

**Feature Envy Results.** The heuristic for *Feature Envy* removal was the most successful one in the experiment we executed. Only two developers gave a *negative* answer, while 12 answers were *intermediate*, and 46 were *positive.* These results gave us confidence about the removal patterns we found. We were able to derive a heuristic that was able to remove the smell very often, according to the developers. The developer we quote below is one of the *intermediate* answer.

> "I believe the refactorings improved the code, but its readability is still not perfect. I believe the constructor is still large." – *Developer with 4 years of experience*

In this case, the developer agrees the heuristic was positive, but there is a suggestion to keep improving the source code. Most of the *intermediate* answers mention some other possible improvements, i.e., the developers think more refactorings are needed to reach an ideal state. However, most of those improvements were not closely related to the purpose of the heuristic being evaluated. Nevertheless, the most experienced developer that participated in the experiment said:

"I agree with the proposed refactorings. They were enough to remove the code smell." – *Developer with 14 years of experience.*

In summary, the vast majority of the answers were positive for all evaluated samples. We achieved a rate of 96.7% of success on the code structure improvement, as pointed out by the developers. We only got two *negative* answers.

**God Class Results.** The derived heuristic we presented in Section 5.3.2 suggests we can remove a *God Class* by recommending several method-moving refactorings. In the three batches we presented to the developers, we recommended several *Move Methods* to remove the *God Class*. All three classes are very large and contains thousands of lines and dozens of methods. Even in these highly complicated scenarios, the heuristics achieved 12 (20%) of *intermediate* answers, and 44 (73.3%) of *positive answers*. Interestingly, not even a single one developer criticized a proposed *Move Method*. There was no complain regarding the suggested refactorings. All complains were related to the continuity of the improvements, i.e., the developers were expecting more *Move Methods* to solve the problem, as the one we quote below.

"This class still needs more refactorings, because I think it still contains several responsibilities." – *Developer with 1 year of experience.*

These results are exciting because it shows a difference between what we consider *God Class*, and what some developers think. According to our *God Class* detection rules, the target classes were not affected by the smell anymore after the refactorings, while the developers still think it is. In this case, we could change the rule to be more severe on the *God Class* detection. For instance, we can reduce the threshold of the number of lines a class must have to be classified as *God Class*. If we change the threshold, the heuristic would continue to suggest *Move Methods*, and the unsatisfied developers might be satisfied if we use the new rule. However, even when recommending several refactorings, most of developers agree the outcome is positive, as the one we quote below:

"I believe the class was very confusing before the refactorings. Now, after the refactorings, the class is way easier to understand and maintain." – *Developer with 2 years of experience.*

Section 6.2.1 presents the goal and research question of the study of this chapter. Our objective was to answer the research question: "Are the smell-removing heuristics able to improve the code structural quality?”. After analyzing the results of our quasi-experiment, we can say the answer to this research question is *yes*. Even with some preliminary heuristics, we were able to achieve a high acceptance of the developers. In this way, it seems worthwhile to follow the path of improving and creating more heuristics.

## 6.3 Threats to Validity

This section presents some threats that could limit the validity of our main findings. For each threat, we present the actions taken to mitigate their impact on the research results. The first threat to validity is related to the number of participants in the study. We have selected a sample of 20 participants, which may not be enough to achieve conclusive results. Additionally, the difference between the developers’ background knowledge can also be a threat. However, as a preliminary study this setting seems to be reliable. We were able to achieve unexpected good results during this study.

The second threat is related to possible misunderstandings during the study. As we asked developers to evaluate code transformations and to answer a questionnaire, they could have conducted the study different from what we asked. To mitigate this threat, we wrote a thorough instructions web page and encouraged them to reach us in case of any doubt. We highlighted that our help would be limited to only clarify the study in order to avoid some bias in our results.

Finally, there is a threat concerning the selected classes and batches. The first one is about the difficulty of the participants in understanding the source code used in the experimental tasks. To mitigate such threat, we described each class thoroughly, and we were very careful to explain what was going on on each step of the generated batches.

## 6.4 Summary

In this chapter, we present a study that aimed at evaluating the heuristics described in Section 6.1. Towards their evaluation, we executed and reported a quasi-experiment in this very chapter. The presented results show the heuristics are promising, leading to interesting and well-evaluated recommendations. However, we only proposed and evaluated three heuristics, so this is only the first step towards a possible recommendation system.

Although we got a high number *positive* answers, we still got a reasonable number of *intermediate* ones (20% of all the answers). It is worthwhile to work towards the reduction of this number since we could increase the satisfaction of the developers. In order to do this, we can explore more of the removal batch-smell patterns presented in Section 5.3. We can derive more heuristics to remove the same code smells presented in this chapter, and run them in the smelly elements used during this quasi-experiment. In this way, the quasi-experiment results can serve as an oracle to test the newly derived heuristics, i.e., we would have an easy way to check if the new heuristics solved the problems pointed out by the developers in our oracle.

# 7 Conclusion

Developers constantly have to improve the internal structure of software systems to satisfy evolving requirements. For this purpose, refactoring has been one of the most common activities applied by developers during software maintenance and evolution [1]. Examples of common refactoring types include [1]: (i) restructuring or moving class members, such as *Extract Method*, *Move Method* and *Pull Up Method*, and (ii) extracting new elements, such as *Extract Superclass* and *Extract Interface*. Regardless the refactoring type, developers need to know when they should refactor the source code; more specifically, what code elements (packages, classes, methods, and the like) need to be refactored. To this end, developers can monitor indicators of poor structural quality in source code to identify opportunities to refactor, such as code smells.

Although refactoring and code smells are strongly related to each other [3], we have little understanding about characteristics concerning this relationship. For instance, we do not know if refactorings applied by developers affect the density of smells in practice (Research Problem 1). One might expect that refactorings reduce the number of code smells. However, such expectation may not happen in practice, specially considering that refactoring is an error-prone activity. Thus, a refactoring may introduce smells rather than remove them. Indeed, it is reasonable to assume that refactorings have strong potential for removing but also for introducing code smells. Unfortunately, we still do not know when refactoring introduces, neglects or removes code smells in practice (Research Problem 2).

Indeed, empirical knowledge about the smell and refactoring is limited. Although they both have been the focus of several studies [11, 13, 17, 18], we do not know to what extent each refactoring interferes in the presence of smells. As a matter of fact, our lack of knowledge is even more evident when we take into account batch refactoring, which is a sequence of refactorings applied to the same element. Sometimes a single refactoring is not enough for developers accomplish their goal; thus, they have to apply other refactorings. Even to remove a single smell, a developer may need to apply multiple refactorings to remove it completely. Despite batch refactorings are constantly applied by developers in practice [1], these mentioned studies tend to ignore them. Actually, we do not know how to identify when a batch refactoring happened (Research Problem 3). Assuming that we are able to identify batch refactoring, we do not know if results for single refactoring (Research Problems 1 and 2) maintain in the context of batch refactoring. Consequently, we also need to investigate whether and when batches introduce, neglect, or remove smells (Research Problem 4). In summary, investigations towards the understanding of how refactorings applied by developers affect the existence of smells is crucial to improve the refactoring practice in the context of smell creation and removal (General Problem). To achieve such a goal, we conducted several studies and proposed some techniques, which are summarized as follows.

## 7.1 Revisiting the Thesis Contributions

In our quest to understand how developers refactor in practice, our first step was to investigate how refactorings affect the density of smells in software projects. Thus, we conducted a longitudinal retrospective study (Chapter 3) to answer the following research question: *Does refactoring reduce the density of code smells?* To answer this research question, we analyzed the source code of 23 software projects. In this procedure, we collected refactorings and code smells in each system. Then, we analyzed the impact of each refactoring on code smells considering only their scope (Section 2.1.4). In our study, we classify a given refactoring instance in one of the three cases: (i) *positive* if the absolute number of smells in the elements decreases after the refactoring; (ii) *negative* if it increases; or (iii) *neutral* if it remains the same. This classification is used to analyze whether certain refactoring types tend to improve or decrease the smelly structure of a program. As a result of this analysis, we found that most refactorings tend to not affect the presence of smells. In this context, the first contribution of this thesis was:

> **1st Contribution.** A first evidence that refactoring operations usually do not remove code smells. On the contrary, they often introduce new ones. Even when developers use the root-canal refactoring tactic, they often degrade the structural quality rather than improve it.

Even though our results indicate that in most cases refactorings do not change the density of code smells, we found several scenarios at which specific types of refactorings affected particular code smell types. For instance, we found several cases of *Extract Superclass* refactoring introducing the *Speculative Generality* code smell. Unexpected relations like that led us to perform a deeper investigation into this kind of relation between refactoring and code smell types.

In this follow-up investigation, we aimed at answering the following research question: *What are the patterns governing types of refactoring and code smells?* To answer this research question, we investigated how each refactoring type interfered on each code smell type. In order to study such interference, we considered three different refactoring-smell patterns: non-removal, removal, and creational (Section 2.4). We were able to identify some examples of these refactoring-smell patterns. For instance, we observed that harmful patterns are frequent, including: (i) approximately 30% of the *Move Method* and *Pull Up Method* refactorings induced the emergence of *God Class*, and (ii) the *Extract Superclass* refactoring creates the smell in 68% of the cases. In this context, the second contribution was:

**2nd Contribution.** There are non-removal, removal, and creational patterns. Developers apply specific refactoring types on code affected by specific smells consistently. Still, they are not able to remove them frequently. In fact, they consistently introduce smells after refactoring.

These first two contributions provide knowledge about the relation between smells and refactoring. This knowledge can be used to improve the techniques to support developers during refactoring. For instance, a tool can benefit from these patterns to aware developers when they perform refactorings.

As discussed before, the first two research questions considered only single refactorings. However, developers often apply refactorings in a sequence [1]. In this way, the findings reported after answering the first two research questions could be just a narrow view of the refactoring practice. For instance, we reported that 33.3% of the single refactorings are negative, but we ignored the fact those refactorings could be part of a batch, and if we considered the entire sequence of refactorings, a given negative single refactoring could be part of a positive batch. If this were the case most of the times, the impact of negative single refactorings could be just ignored.

In order to check the validity of the first findings when considering batch refactorings, we conducted a different study. In this study, we aimed at answering the following research question: *Does batch refactoring impact the density of code smells?* To answer this research question, we first had to learn how to detect a batch of refactorings when looking at a sea of single refactorings (Section 4.2). Once we synthesized the batch refactorings, we were able to classify each one according to their impact on the existence of code smells (Section 4.3). In this vein, we were able to check if the code smells are affected differently if we consider the batch and not only the single refactorings. Surprisingly, batch refactorings tend to follow the same trends that we revealed when studying single refactorings. After comparing the classification results of batch and single refactorings, we were able to report our third contribution:

**3rd Contribution.** A first evidence that either single or batch refactorings often do not reduce the density of code smells. In fact, most of the batches are also neutral or negative.

After concluding that batch refactorings often do not reduce the density of smells – which is similar to the behavior of single refactorings – we can advance our research by investigating the existence of batch-smell patterns. In this step, we aim at answering the following research question: *What are the patterns governing batches and code smells?* To answer this research question, we followed a similar methodology to the one used to answer $RQ_2$ (Section 3.1.1). In order to detect batch-smell patterns, we observed how each instance of batch types affected each code smell type. We were able to detect several batch-smell patterns that can be used to derive heuristics of how developers can remove code smells. Therefore, the fourth contribution of this thesis was:

**4th Contribution.** There are various removal and creational batch-smell patterns. By analyzing multiple patterns, we were able to derive a first suite of heuristics that developers can use to remove code smells through batch refactorings.

The batch-smell patterns explain how developers apply batches in practice and how they affect different types of code smells. These patterns give us an unreported facet of the refactoring practice. From them, we can better understand how developers behave in the presence of code smells in the context of refactoring. By better understanding these behaviors, we can learn and report guidelines of how developers can refactor different types of code smells, as reported in Section 6.1. Based on this knowledge, we describe three preliminary heuristics to remove *Feature Envy*, *God Class*, and *Complex Class*. We evaluated these heuristics with 20 developers and the results were promising. More than 90% of the developers' answers indicate that the heuristics were able to, somehow, improve the code structure (Section 6.2).

We highlight that before proposing solutions that will help developers to better refactor, first, we need to understand how they conduct refactoring in practice. Therefore, the contributions listed here can advance both state-of-the-art and state-of-the-practice. Researchers and industry practitioners can use the discussions in this thesis to define mechanisms that drive news solutions for supporting developers during refactoring operations. The knowledge provided here can help researchers in building the most suitable mechanisms that are aligned with how developers perform refactoring in practice.

Finally, to facilitate future references to works that resulted from this thesis, Table 7.1 presents the papers produced in the context of this thesis and the respective chapters to which they are related.

Table 7.1: Papers produced in the context of this thesis

| Chapters | Paper |
|---|---|
| 2–3 | **Diego Cedrim**, Leonardo Sousa, Alessandro Garcia, and Rohit Gheyi. 2016. *Does refactoring improve software structural quality? A longitudinal study of 25 projects.* In Proceedings of the 30th Brazilian Symposium on Software Engineering (SBES '16) |
| 2–3 | **Diego Cedrim**. 2016. *Context-sensitive identification of refactoring opportunities.* In Proceedings of the 38th International Conference on Software Engineering Companion – Doctoral Symposium (ICSE '16). ACM, New York, NY, USA, 827-830. |
| 2–3 | Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, **Diego Cedrim**, and Alessandro Garcia. 2017. *How does refactoring affect internal quality attributes?: A multi-project study.* In Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES'17). ACM, New York, NY, USA, 74-83. |
| 2–3 | **Diego Cedrim**, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. *Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects.* In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017). ACM, New York, NY, USA, 465-475. |
| 2–3 | Leonardo Sousa, Isabella Ferreira, **Diego Cedrim**, Alexander Chávez, Alessandro Garcia, and Carlos Lucena. 2018. *The Structural Quality of Refactored Code: A Study of 50 Software Projects.* Journal. Under submission process. |
| 4–6 | **Diego Cedrim**, Alessandro Garcia, Leonardo Sousa, Anderson Oliveira, Ana Bibiano, Isabela Vieira. 2018. *Batch Refactorings at a Glance.* IEEE Transactions on Software Engineering. Under submission process. |
| 5 | Ana Bibiano, **Diego Cedrim**, Eduardo Fernandes, Daniel Tenório, Isabella Ferreira, Anderson Oliveira, Roberto Oliveira, Alessandro Garcia, Baldoino Fonseca, Marcos Kalinowski. 2018. *Batch Refactorings at a Glance. Exploring the Effects of Batch Refactoring on Program Maintainability: A Heuristic-based Study.* ICSE-SEIP'19. Under submission process. |

## 7.2 Future Work

New challenges and opportunities for improvement have emerged along the studies conducted in the context of this thesis. Based on them, further directions for future works are presented next.

**Recommender System.** We presented in Section 6.1 three possible heuristics to remove *God Class*, *Feature Envy*, and *Complex Class* instances. Although we executed a preliminary validation of them, we did not develop a fully-fledged recommender system. An ideal solution would be a system that is capable of evaluating code smells in real time as a plug-in of any major Integrated Development Environment (IDE). In this way, developers would benefit from such heuristics and could remove code smells in a guided interactive way. Our preliminary solution is already capable of detecting code smells in real time, and it is available publicly under MIT License at `https://github.com/opus-research/refresh`. Anyone can fork and proceed with the remaining development of this tool.

**New Heuristics.** Although we presented three heuristics, there are a plethora of unreported ones to derive. Even for *God Class*, *Feature Envy*, and *Complex Class*, we can propose several different heuristics to remove them. A new one can be more suitable in a different development contexts or programming languages. Besides that, one might also use our data to propose heuristics to different code smells, such as *Long Method*, *Brain Class*, and *Lazy Class*. Those new potential heuristics can be gradually integrated into the recommender system mentioned before. In this way, developers can have a better support when refactoring code smells, and, hopefully, the frequency of negative refactorings can ease over time.

**Investigating Heuristics to Prioritize Smells.** A software system can have thousands of code smells [43], which can lead to thousands of recommendations of refactorings. However, not all code smells are related to relevant maintainability problems, so not all smells urge for refactoring. Thus, in order to support developers during the refactoring process, we need to investigate heuristics to prioritize smells that are most likely to indicate a problem that is severe for the system maintainability. Only after this, we should generate refactoring recommendations. The current version of our tool implements a rudimentary prioritization algorithm. We used this prioritization in order to extract the examples we used to validate our heuristics (Section 6.2).

**Continuous Integration.** Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early [82]. In this context, an interesting improvement would be integrating our tool to the Continuous Integration cycle. In this way, we would be able to detect negative refactorings at commit time, immediately after the code being pushed to the shared repository. This mechanism could help development teams to spot erroneous refactorings early and work to solve the problem as soon as possible.

# Bibliography

[1] MURPHY-HILL, E.; PARNIN, C. ; BLACK, A. P.. **How We Refactor, and How We Know It**. IEEE Transactions on Software Engineering, 38(1):5–18, 2012.

[2] STROGGYLOS, K.; SPINELLIS, D.. **Refactoring - Does it Improve Software Quality?** In: PROCEEDINGS OF THE 5TH INTERNATIONAL WORKSHOP ON SOFTWARE QUALITY, 2007.

[3] FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W. ; ROBERTS, D.. **Refactoring: Improving The Design Of Existing Code**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[4] TOURWÉ, T.; MENS, T.. **Identifying Refactoring Opportunities Using Logic Meta Programming**. In: PROCEEDINGS OF THE 7TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, CSMR '03, p. 91–100, Washington, DC, USA, 2003. IEEE Computer Society.

[5] MACIA, I.; ARCOVERDE, R.; GARCIA, A.; CHAVEZ, C. ; VON STAA, A.. **On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms**. Proceedings of the 16th European Conference on Software Maintenance and Reengineering, p. 277–286, 2012.

[6] YAMASHITA, A.; MOONEN, L.. **Do Developers Care About Code Smells? An Exploratory Survey.** In: Lommel, R.; Oliveto, R. ; Robbes, R., editors, PROCEEDINGS OF THE 20TH WORKING CONFERENCE ON REVERSE ENGINEERING, p. 242–251. IEEE Computer Society, 2013.

[7] PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R. ; LUCIA, A. D.. **Do They Really Smell Bad? A Study On Developers' Perception of Bad Code Smells**. In: PROCEEDINGS OF THE 30TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, p. 101–110, 2014.

[8] BOURQUIN, F.; KELLER, R. K.. **High-Impact Refactoring Based on Architecture Violations**. In: PROCEEDINGS OF THE 11TH EU-

ROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, p. 149–158, 2007.

[9] OIZUMI, W.; GARCIA, A.; DA SILVA SOUSA, L.; CAFEO, B. ; ZHAO, Y.. **Code Anomalies Flock Together: Exploring Code Anomaly Agglomerations for Locating Design Problems.** In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '16, p. 440–451, New York, NY, USA, 2016. ACM.

[10] YAMASHITA, A.. **Assessing the Capability of Code Smells to Explain Maintenance Problems: An Empirical Study Combining Quantitative And Qualitative Data.** Empirical Software Engineering, 19(4):1111–1143, 2013.

[11] RATZINGER, J.; SIGMUND, T. ; GALL, H. C.. **On The Relation of Refactorings and Software Defect Prediction.** In: PROCEEDINGS OF THE INTERNATIONAL WORKSHOP ON MINING SOFTWARE REPOSITORIES, p. 35–38, New York, New York, USA, 2008. ACM Press.

[12] KHOMH, F.; PENTA, M. D.; GUÉHÉNEUC, Y.-G. ; ANTONIOL, G.. **An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness.** Empirical Software Engineering, 17(3):243–275, 2012.

[13] FUJIWARA, K.; FUSHIDA, K.; YOSHIDA, N. ; IIDA, H.. **Assessing Refactoring Instances and the Maintainability Benefits of Them from Version Archives**, p. 313–323. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[14] YAMASHITA, A.; COUNSELL, S.. **Code Smells as System-Level Indicators of Maintainability: An Empirical Study.** Journal of Systems and Software, 86(10):2639–2653, 2013.

[15] SILVA, D.; TSANTALIS, N. ; VALENTE, M. T.. **Why We Refactor? Confessions of GitHub Contributors.** In: PROCEEDINGS OF THE 24TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, FSE 2016, p. 858–870, New York, NY, USA, 2016. ACM.

[16] MURPHY, G.; KERSTEN, M. ; FINDLATER, L.. **How Are Java Software Developers Using The Elipse IDE?** IEEE Software, 23(4):76–83, 2006.

[17] BAVOTA, G.; DE LUCIA, A.; DI PENTA, M.; OLIVETO, R. ; PALOMBA, F.. **An Experimental Investigation On The Innate Relationship**

**Between Quality And Refactoring**. Journal of Systems and Software, 107:1–14, 2015.

[18] SJOBERG, D. I. K.; YAMASHITA, A.; ANDA, B. C. D.; MOCKUS, A. ; DYBA, T.. **Quantifying The Effect Of Code Smells On Maintenance Effort**. IEEE Transactions on Software Engineering, 39(8):1144–1156, 2013.

[19] PAIXÃO, M.. **Software Restructuring: Understanding Longitudinal Architectural Changes and Refactoring**. PhD thesis, University College London, 2018.

[20] SANTOS, P.; TRAVASSOS, G.. **Chapter 5 - Action Research Can Swing The Balance In Experimental Software Engineering**. volumen 83 de **Advances in Computers**, p. 205–276. Elsevier, 2011.

[21] BAVOTA, G.; DE CARLUCCIO, B.; DE LUCIA, A.; DI PENTA, M.; OLIVETO, R. ; STROLLO, O.. **When Does a Refactoring Induce Bugs? An Empirical Study**. Proceedings of the IEEE 12th International Working Conference on Source Code Analysis and Manipulation, p. 104–113, 2012.

[22] KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **An Empirical Study of Refactoring Challenges and Benefits at Microsoft**. IEEE Transactions on Software Engineering, 40(7):633–649, 2014.

[23] ŚLIWERSKI, J.; ZIMMERMANN, T. ; ZELLER, A.. **When do Changes Induce Fixes?** ACM SIGSOFT Software Engineering Notes, 30(4):1–5, 2005.

[24] TABA, S. E. S.; KHOMH, F.; ZOU, Y.; HASSAN, A. E. ; NAGAPPAN, M.. **Predicting Bugs Using Antipatterns**. In: PROCEEDINGS OF THE IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, p. 270–279, 2013.

[25] PALOMBA, F.; ZANONI, M.; FONTANA, F. A.; LUCIA, A. D. ; OLIVETO, R.. **Smells Like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells**. In: PROCEEDINGS OF THE IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, p. 244–255, 2016.

[26] KIM, S.; ZIMMERMANN, T.; PAN, K. ; WHITEHEAD, E. J. J.. **Automatic Identification of Bug-Introducing Changes**. In: PROCEED-

INGS OF THE 21ST IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, p. 81–90, 2006.

[27] PRETE, K.; RACHATASUMRIT, N.; SUDAN, N. ; KIM, M.. **Template-Based Reconstruction of Complex Refactorings**. In: PROCEEDINGS OF IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, p. 1–10, 2010.

[28] TSANTALIS, N.; GUANA, V.; STROULIA, E. ; HINDLE, A.. **A Multidimensional Empirical Study on Refactoring Activity**. In: PROCEEDINGS OF THE CONFERENCE OF THE CENTER FOR ADVANCED STUDIES ON COLLABORATIVE RESEARCH, CASCON '13, p. 132–146, Riverton, NJ, USA, 2013. IBM Corp.

[29] KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **A Field Study of Refactoring Challenges and Benefits**. In: PROCEEDINGS OF THE ACM SIGSOFT 20TH INTERNATIONAL SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, FSE '12, p. 1–11, New York, NY, USA, 2012. ACM.

[30] KOLB, R.; MUTHIG, D.; PATZKE, T. ; YAMAUCHI, K.. **A Case Study in Refactoring a Legacy Component for Reuse in a Product Line**. In: PROCEEDINGS OF THE 21ST IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, p. 369–378, 2005.

[31] MOSER, R.; SILLITTI, A.; ABRAHAMSSON, P. ; SUCCI, G.. **Does Refactoring Improve Reusability?** In: PROCEEDINGS OF THE 9TH INTERNATIONAL CONFERENCE ON REUSE OF OFF-THE-SHELF COMPONENTS, ICSR'06, p. 287–297, Berlin, Heidelberg, 2006. Springer-Verlag.

[32] MACIA, I.; GARCIA, A. ; VON STAA, A.. **An Exploratory Study of Code Smells in Evolving Aspect-Oriented Systems**. In: PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, AOSD '11, p. 203–214, New York, NY, USA, 2011. ACM.

[33] GURGEL, A.; MACIA, I.; GARCIA, A.; VON STAA, A.; MEZINI, M.; EICHBERG, M. ; MITSCHKE, R.. **Blending and Reusing Rules for Architectural Degradation Prevention**. In: PROCEEDINGS OF THE 13TH INTERNATIONAL CONFERENCE ON MODULARITY, MODULARITY '14, p. 61–72, New York, NY, USA, 2014. ACM.

[34] OIZUMI, W. N.; GARCIA, A. F.; COLANZI, T. E.; FERREIRA, M. ; VON STAA, A.. **On the Relationship of Code-Anomaly Agglomerations and Architectural Problems**. Journal of Software Engineering Research and Development, 3(1):11, 2015.

[35] MARTIN, R. C.; MARTIN, M.. **Agile Principles, Patterns, and Practices in C#**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[36] LANZA, M.; MARINESCU, R.. **Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems**. Springer Publishing Company, Incorporated, 1st edition, 2010.

[37] MARINESCU. **Detection Strategies: Metrics-Based Rules for Detecting Design Flaws**. In: PROCEEDINGS OF 20TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, p. 350–359, 2004.

[38] MOHA, N.; GUEHENEUC, Y.; DUCHIEN, L. ; MEUR, A. L.. **Decor: A Method for the Specification and Detection of Code and Design Smells**. IEEE Transaction on Software Engineering, 36:20–36, 2010.

[39] MAIGA, A.; ALI, N.; BHATTACHARYA, N.; SABANÉ, A.; GUÉHÉNEUC, Y.-G.; ANTONIOL, G. ; AÏMEUR, E.. **Support Vector Machines for Anti-Pattern Detection**. In: PROCEEDINGS OF THE 27TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, ASE 2012, p. 278–281, New York, NY, USA, 2012. ACM.

[40] KHOMH, F.; DI PENTA, M. ; GUEHENEUC, Y.-G.. **An Exploratory Study of the Impact of Code Smells on Software Change-Proneness**. Proceedings of the 16th Working Conference on Reverse Engineering, p. 75–84, 2009.

[41] OUNI, A.; GAIKOVINA KULA, R.; KESSENTINI, M. ; INOUE, K.. **Web Service Antipatterns Detection Using Genetic Programming**. In: PROCEEDINGS OF THE 2015 ANNUAL CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, GECCO '15, p. 1351–1358, New York, NY, USA, 2015. ACM.

[42] MACIA, I.; ARCOVERDE, R.; CIRILO, E.; GARCIA, A. ; VON STAA, A.. **Supporting the Identification of Architecturally-Relevant Code Anomalies**. In: PROCEEDINGS OF THE 28TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, p. 662–665, 2012.

[43] MACIA, I.; GARCIA, J.; POPESCU, D.; GARCIA, A.; MEDVIDOVIC, N. ; VON STAA, A.. **Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity? An Exploratory Analysis of Evolving Systems**. In: PROCEEDINGS OF THE 11TH ANNUAL INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, AOSD '12, p. 167–178, New York, NY, USA, 2012. ACM.

[44] HOZANO, M.; GARCIA, A.; ANTUNES, N.; FONSECA, B. ; COSTA, E.. **Smells are Sensitive to Developers! On the Efficiency ff (Un)Guided Customized Detection**. In: PROCEEDINGS OF THE 25TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, ICPC '17, p. 110–120, Piscataway, NJ, USA, 2017. IEEE Press.

[45] CHÁVEZ, A.; FERREIRA, I.; FERNANDES, E.; CEDRIM, D. ; GARCIA, A.. **How Does Refactoring Affect Internal Quality Attributes? A Multi-Project Study**. In: PROCEEDINGS OF THE 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, SBES'17, p. 74–83, New York, NY, USA, 2017. ACM.

[46] ARCOVERDE, R.; MACIA, I.; GARCIA, A. ; VON STAA, A.. **Automatically Detecting Architecturally-Relevant Code Anomalies**. Proceedings of the International Workshop on Recommendation Systems for Software Engineering, p. 90–91, 2012.

[47] MACIA, I.; GARCIA, A.; CHAVEZ, C. ; VON STAA, A.. **Enhancing the Detection of Code Anomalies with Architecture-Sensitive Strategies**. In: PROCEEDINGS OF THE 17TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, p. 177–186. IEEE, 2013.

[48] SOUSA, L.; OLIVEIRA, R.; GARCIA, A.; LEE, J.; CONTE, T.; OIZUMI, W.; DE MELLO, R.; LOPES, A.; VALENTIM, N.; OLIVEIRA, E. ; LUCENA, C.. **How do Software Developers Identify Design Problems? A Qualitative Analysis**. In: PROCEEDINGS OF THE 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, SBES'17, p. 54–63, New York, NY, USA, 2017. ACM.

[49] YAMASHITA, A.; MOONEN, L.. **Exploring the Impact of Inter-Smell Relations on Software Maintainability: An Empirical Study**. Proceedings of the International Conference on Software Engineering, p. 682–691, 2013.

[50] YAMASHITA, A.; MOONEN, L.. **To What Extent can Maintenance Problems be Predicted by Code Smell Detection? An Empirical Study**. Information and Software Technology, 55(12):2223–2242, 2013.

[51] TUFANO, M.; PALOMBA, F.; BAVOTA, G.; OLIVETO, R.; DI PENTA, M.; DE LUCIA, A. ; POSHYVANYK, D.. **When and Why Your Code Starts to Smell Bad**. In: PROCEEDINGS OF THE 37TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '15, p. 403–414, Piscataway, NJ, USA, 2015. IEEE Press.

[52] GE, X.; DUBOSE, Q. L. ; MURPHY-HILL, E.. **Reconciling Manual And Automatic Refactoring**. In: PROCEEDINGS OF THE 34TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 211–221, 2012.

[53] WANG, Y.. **What Motivate Software Engineers to Refactor Source Code? Evidences From Professional Developers**. Proceedings of the IEEE International Conference on Software Maintenance, p. 413–416, 2009.

[54] SOARES, G.; GHEYI, R. ; MASSONI, T.. **Automated Behavioral Testing of Refactoring Engines**. IEEE Transactions on Software Engineering, 39(2):147–162, 2013.

[55] SOARES, G.; GHEYI, R.; MURPHY-HILL, E. ; JOHNSON, B.. **Comparing Approaches to Analyze Refactoring Activity on Software Repositories**. Journal of Systems and Software, 86(4):1006–1022, 2013.

[56] TSANTALIS, N.; CHATZIGEORGIOU, A.. **Identification of Move Method Refactoring Opportunities**. IEEE Transactions on Software Engineering, 35(3):347–367, 2009.

[57] TSANTALIS, N.; CHATZIGEORGIOU, A.. **Identification of Extract Method Refactoring Opportunities**. In: PROCEEDINGS OF THE 13TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, volumen 35, p. 119–128. IEEE, 2009.

[58] TSANTALIS, N.; CHATZIGEORGIOU, A.. **Identification of Refactoring Opportunities Introducing Polymorphism**. Journal of Systems and Software, 83(3):391–404, 2010.

[59] TSANTALIS, N.; CHATZIGEORGIOU, A.. **Identification of Extract Method Refactoring Opportunities for the Decomposition of Methods**. Journal of Systems and Software, 84(10):1757–1782, 2011.

[60] AL DALLAL, J.. **Constructing Models for Predicting Extract Subclass Refactoring Opportunities Using Object-Oriented Quality Metrics**. Information and Software Technology, 54(10):1125–1141, 2012.

[61] BAVOTA, G.; OLIVETO, R.; DE LUCIA, A.; ANTONIOL, G. ; GUEHENEUC, Y.-G.. **Playing With Refactoring: Identifying Extract Class Opportunities Through Game Theory**. Proceedings of the IEEE International Conference on Software Maintenance, p. 1–5, 2010.

[62] BAVOTA, G.; DE LUCIA, A. ; OLIVETO, R.. **Identifying Extract Class Refactoring Opportunities Using Structural and Semantic Cohesion Measures**. Journal of Systems and Software, 84(3):397–414, 2011.

[63] PAPPALARDO, G.; TRAMONTANA, E.. **Suggesting Extract Class Refactoring Opportunities by Measuring Strength of Method Interactions**. Proceedings of the Asia-Pacific Software Engineering Conference, 2:105–110, 2013.

[64] BOIS, B. D.; DEMEYER, S. ; VERELST, J.. **Refactoring - Improving Coupling And Cohesion Of Existing Code**. Proceedings of the Working Conference on Reverse Engineering, p. 144–151, 2004.

[65] MEANANEATRA, P.. **Identifying Refactoring Sequences For Improving Software Maintainability**. In: PROCEEDINGS OF THE 27TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, p. 406–409, New York, New York, USA, 2012. ACM Press.

[66] DIETRICH, J.; MCCARTIN, C.; TEMPERO, E. ; SHAH, S. M. A.. **On the Existence of High-impact Refactoring Opportunities in Programs**. In: PROCEEDINGS OF THE AUSTRALASIAN COMPUTER SCIENCE CONFERENCE, ACSC '12, p. 37–48, Darlinghurst, Australia, Australia, 2012. Australian Computer Society, Inc.

[67] HOTTA, K.; HIGO, Y. ; KUSUMOTO, S.. **Identifying, Tailoring, and Suggesting Form Template Method Refactoring Opportunities with Program Dependence Graph**. Proceedings of the 16th European Conference on Software Maintenance and Reengineering, p. 53–62, 2012.

[68] MELTON, H.; TEMPERO, E.. **Identifying Refactoring Opportunities by Identifying Dependency Cycles**. p. 35–41, 2006.

[69] TSANTALIS, N.. **Refactoring Miner GitHub Page**. `https://github.com/tsantalis/RefactoringMiner`, 2017. [Online; accessed 6-july-2017; version 0.2.0].

[70] XING, Z.; STROULIA, E.. **Umldiff: An Algorithm for Object-Oriented Design Differencing**. In: PROCEEDINGS OF THE 20TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, ASE '05, p. 54–65, New York, NY, USA, 2005. ACM.

[71] MARA, L.; HONORATO, G.; MEDEIROS, F. D.; GARCIA, A. ; LUCENA, C.. **Hist-Inspect: A Tool for History-Sensitive Detection of Code Smells**. In: PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT COMPANION, AOSD '11, p. 65–66, New York, NY, USA, 2011. ACM.

[72] MACIA, I.. **On The Detection Of Architecturally Relevant Code Anomalies In Software Systems**. PhD thesis, Pontifical Catholic University of Rio de Janeiro, 2013.

[73] CEDRIM, D.; SOUSA, L.. **Organic**. `https://github.com/diegocedrim/organic`, 2018. [Online; accessed 20-september-2018].

[74] FERREIRA, M.; BARBOSA, E.; MACIA, I.; ARCOVERDE, R. ; GARCIA, A.. **Detecting Architecturally-Relevant Code Anomalies: A Case Study of Effectiveness and Effort**. In: PROCEEDINGS OF THE 29TH ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, SAC '14, p. 1158–1163, New York, NY, USA, 2014. ACM.

[75] CEDRIM, D.; DA SILVA SOUSA, L.; GARCIA, A. F. ; GHEYI, R.. **Does Refactoring Improve Software Structural Quality? A Longitudinal Study of 25 Projects**. In: PROCEEDINGS OF THE 30TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p. 73–82, New York, NY, USA, 2016. ACM.

[76] ANTONIOL, G.; GUEHENEUC, Y.; MERLO, E. ; TONELLA, P.. **Mining the Lexicon Used by Programmers During Sofware Evolution**. Proceedings of IEEE International Conference on Software Maintenance, p. 14–23, 2007.

[77] KAPSER, C. J.; GODFREY, M. W.. **"Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software**. Empirical Software Engineering, 13(6):645–692, 2008.

[78] BIBIANO, A.. **Exploring the Effects of Batch Refactoring on Program Maintainability: A Heuristic-Based Study**. Technical report, Pontifical Catholic University of Rio de Janeiro, 2018.

[79] CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; DE MELLO, R.; FONSECA, B.; RIBEIRO, M. ; CHÁVEZ, A.. **Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects**. In: PROCEEDINGS OF THE 11TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, ESEC/FSE 2017, p. 465–475, New York, NY, USA, 2017. ACM.

[80] CHATZIGEORGIOU, A.; CHARALAMPIDOU, S.; AMPATZOGLOU, A. ; CHATZIGEORGIOU, A.. **Identifying Extract Method Refactoring Opportunities Based on Functional Relevance**. IEEE Transactions on Software Engineering, 43(July):1–22, 2017.

[81] EASTERBROOK, S.; SINGER, J.; STOREY, M.-A. ; DAMIAN, D.. **Selecting Empirical Methods for Software Engineering Research**, p. 285–311. Springer London, London, 2008.

[82] FOWLER, M.; FOEMMEL, M.. **Continuous Integration**. Thought-Works) http://www.thoughtworks.com/Continuous Integration.pdf, 122:14, 2006.

# A
# Remaining Batch-Smell Patterns

Section 5.3 presents batch-smell patterns related to the smells *Feature Envy*, *God Class*, and *Complex Class*. However, we found additional patterns related to several others code smells, as presented in this appendix.

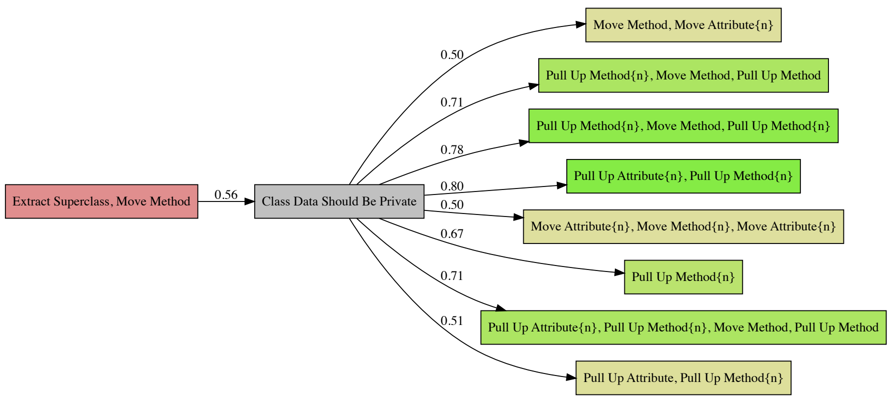

Extract Superclass, Move Method
0.56
Class Data Should Be Private
0.50
Move Method, Move Attribute{n}
0.71
Pull Up Method{n}, Move Method, Pull Up Method
0.78
Pull Up Method{n}, Move Method, Pull Up Method{n}
0.80
Pull Up Attribute{n}, Pull Up Method{n}
0.50
Move Attribute{n}, Move Method{n}, Move Attribute{n}
0.67
Pull Up Method{n}
0.71
Pull Up Attribute{n}, Pull Up Method{n}, Move Method, Pull Up Method
0.51
Pull Up Attribute, Pull Up Method{n}

Figure A.1: Class Data Should be Private patterns

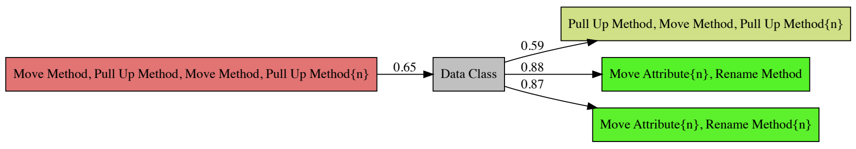

Move Method, Pull Up Method, Move Method, Pull Up Method{n}
0.65
Data Class
0.59
Pull Up Method, Move Method, Pull Up Method{n}
0.88
Move Attribute{n}, Rename Method
0.87
Move Attribute{n}, Rename Method{n}

Figure A.2: Data Class patterns

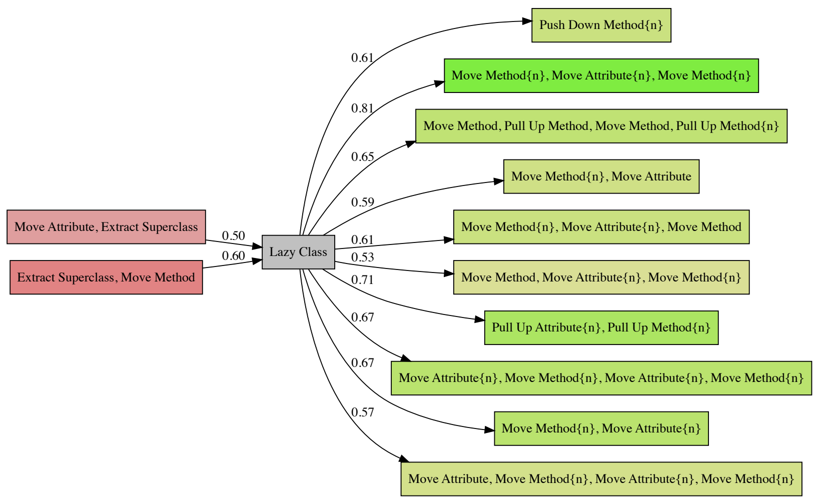

Move Attribute, Extract Superclass
Extract Superclass, Move Method
0.50
0.60
Lazy Class
0.61
0.81
0.65
0.59
0.61
0.53
0.71
0.67
0.67
0.57
Push Down Method{n}
Move Method{n}, Move Attribute{n}, Move Method{n}
Move Method, Pull Up Method, Move Method, Pull Up Method{n}
Move Method{n}, Move Attribute
Move Method{n}, Move Attribute{n}, Move Method
Move Method, Move Attribute{n}, Move Method{n}
Pull Up Attribute{n}, Pull Up Method{n}
Move Attribute{n}, Move Method{n}, Move Attribute{n}, Move Method{n}
Move Method{n}, Move Attribute{n}
Move Attribute, Move Method{n}, Move Attribute{n}, Move Method{n}

Figure A.3: Lazy Class patterns

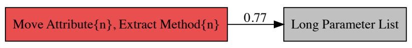


Figure A.4: Long Parameter List pattern

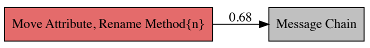


Figure A.5: Message Chain pattern

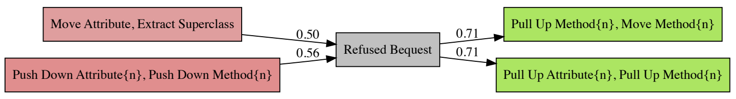

Move Attribute, Extract Superclass
Push Down Attribute{n}, Push Down Method{n}
0.50
0.56
Refused Bequest
0.71
0.71
Pull Up Method{n}, Move Method{n}
Pull Up Attribute{n}, Pull Up Method{n}

Figure A.6: Refused Bequest patterns

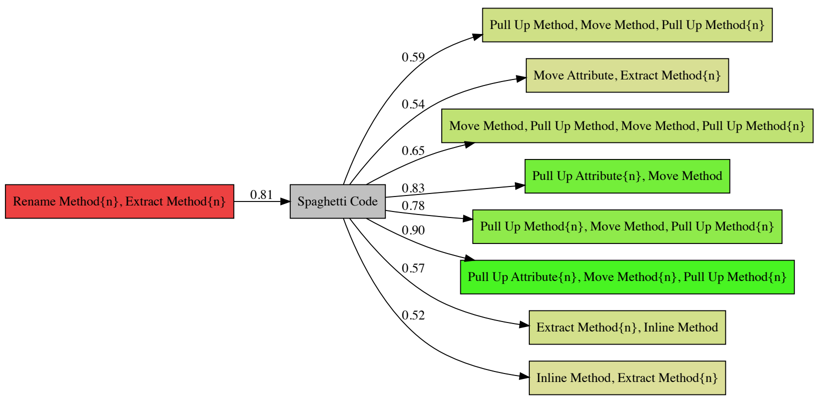

Rename Method{n}, Extract Method{n}
0.81
Spaghetti Code
0.59
0.54
0.65
0.83
0.78
0.90
0.57
0.52
Pull Up Method, Move Method, Pull Up Method{n}
Move Attribute, Extract Method{n}
Move Method, Pull Up Method, Move Method, Pull Up Method{n}
Pull Up Attribute{n}, Move Method
Pull Up Method{n}, Move Method, Pull Up Method{n}
Pull Up Attribute{n}, Move Method{n}, Pull Up Method{n}
Extract Method{n}, Inline Method
Inline Method, Extract Method{n}

Figure A.7: Spaghetti Code patterns

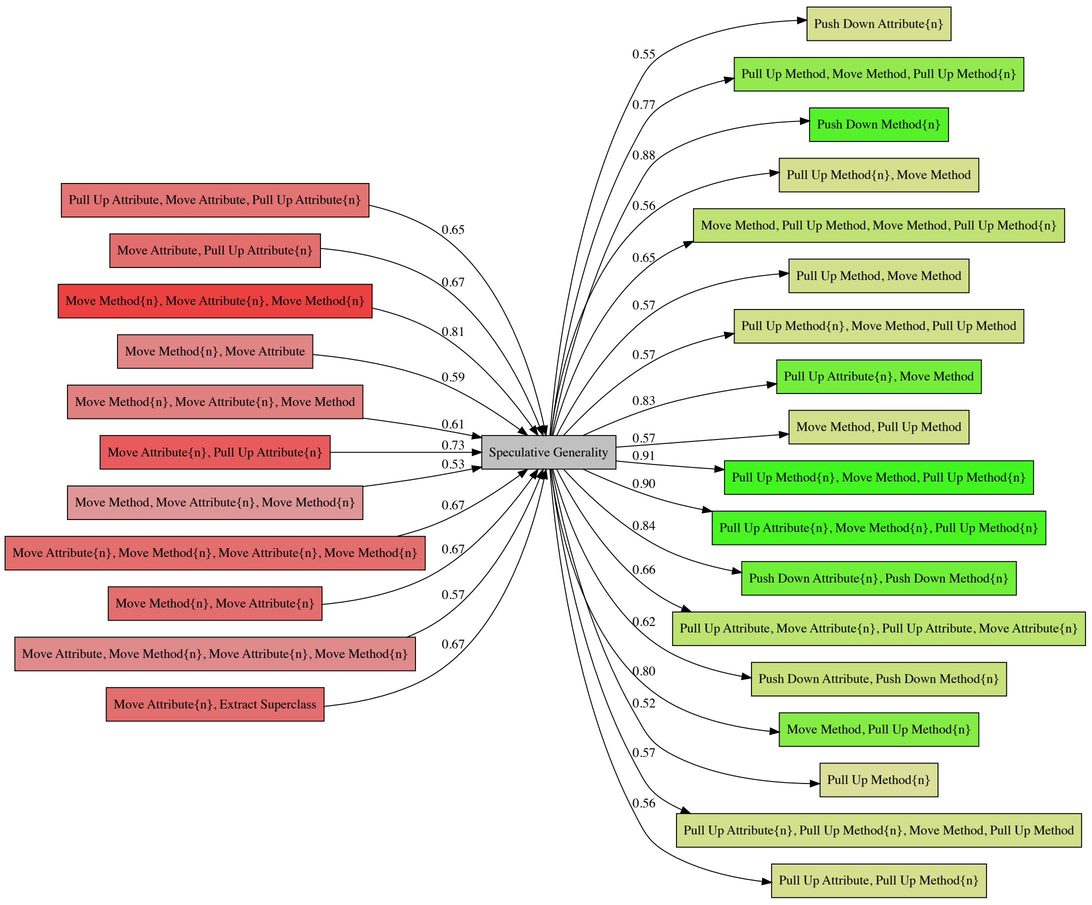
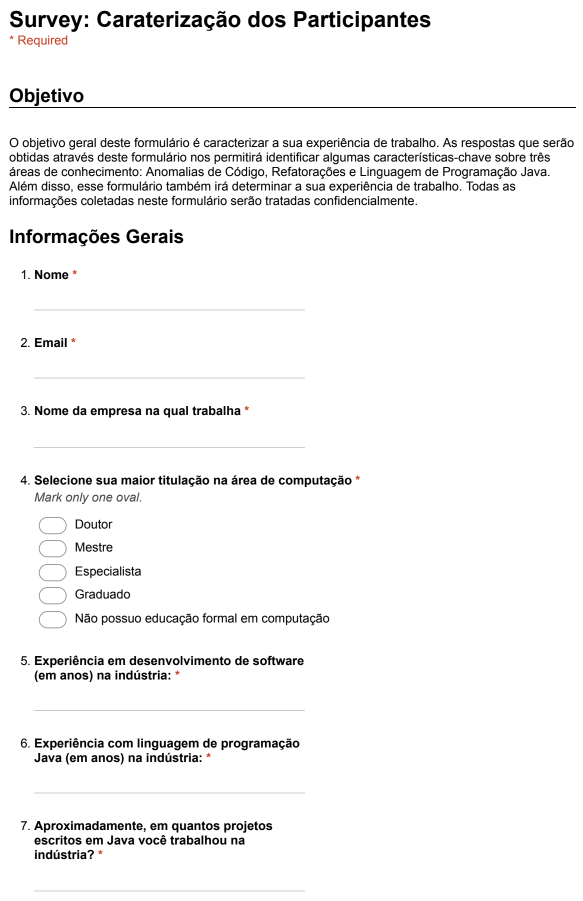

Pull Up Attribute, Move Attribute, Pull Up Attribute{n}
0.65
Move Attribute, Pull Up Attribute{n}
0.67
Move Method{n}, Move Attribute{n}, Move Method{n}
0.81
Move Method{n}, Move Attribute
0.59
Move Method{n}, Move Attribute{n}, Move Method
0.61
Move Attribute{n}, Pull Up Attribute{n}
0.73
0.53
Move Method, Move Attribute{n}, Move Method{n}
0.67
Move Attribute{n}, Move Method{n}, Move Attribute{n}, Move Method{n}
0.67
Move Method{n}, Move Attribute{n}
0.57
Move Attribute, Move Method{n}, Move Attribute{n}, Move Method{n}
0.67
Move Attribute{n}, Extract Superclass
Speculative Generality
0.55
Push Down Attribute{n}
0.77
Pull Up Method, Move Method, Pull Up Method{n}
0.88
Push Down Method{n}
0.56
Pull Up Method{n}, Move Method
0.65
Move Method, Pull Up Method, Move Method, Pull Up Method{n}
0.57
Pull Up Method, Move Method
0.57
Pull Up Method{n}, Move Method, Pull Up Method
0.83
Pull Up Attribute{n}, Move Method
0.57
Move Method, Pull Up Method
0.91
Pull Up Method{n}, Move Method, Pull Up Method{n}
0.90
Pull Up Attribute{n}, Move Method{n}, Pull Up Method{n}
0.84
Push Down Attribute{n}, Push Down Method{n}
0.66
Pull Up Attribute, Move Attribute{n}, Pull Up Attribute, Move Attribute{n}
0.62
Push Down Attribute, Push Down Method{n}
0.80
Move Method, Pull Up Method{n}
0.52
Pull Up Method{n}
0.57
Pull Up Attribute{n}, Pull Up Method{n}, Move Method, Pull Up Method
0.56
Pull Up Attribute, Pull Up Method{n}

Figure A.8: Speculative Generality patterns

# B
# Quasi-Experiment Subject Characterization Questionnaire

This appendix presents the subject characterization questionnaire. This questionnaire was used in the study reported in Chapter 6.



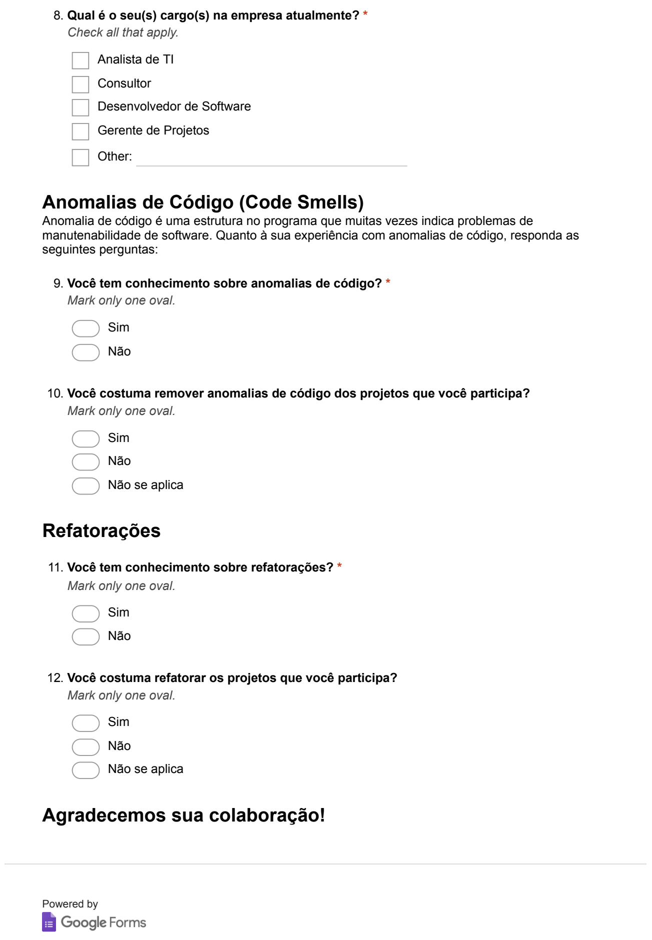
**Survey: Caraterização dos Participantes**
* Required

**Objetivo**

O objetivo geral deste formulário é caracterizar a sua experiência de trabalho. As respostas que serão obtidas através deste formulário nos permitirá identificar algumas características-chave sobre três áreas de conhecimento: Anomalias de Código, Refatorações e Linguagem de Programação Java. Além disso, esse formulário também irá determinar a sua experiência de trabalho. Todas as informações coletadas neste formulário serão tratadas confidencialmente.

**Informações Gerais**

1. **Nome** *
2. **Email** *
3. **Nome da empresa na qual trabalha** *
4. **Selecione sua maior titulação na área de computação** *
   *Mark only one oval.*
   - ◯ Doutor
   - ◯ Mestre
   - ◯ Especialista
   - ◯ Graduado
   - ◯ Não possuo educação formal em computação
5. **Experiência em desenvolvimento de software (em anos) na indústria:** *
6. **Experiência com linguagem de programação Java (em anos) na indústria:** *
7. **Aproximadamente, em quantos projetos escritos em Java você trabalhou na indústria?** *



8. **Qual é o seu(s) cargo(s) na empresa atualmente?** *

*Check all that apply.*

- ☐ Analista de TI
- ☐ Consultor
- ☐ Desenvolvedor de Software
- ☐ Gerente de Projetos
- ☐ Other: ____________

## Anomalias de Código (Code Smells)

Anomalia de código é uma estrutura no programa que muitas vezes indica problemas de manutenabilidade de software. Quanto à sua experiência com anomalias de código, responda as seguintes perguntas:

9. **Você tem conhecimento sobre anomalias de código?** *

*Mark only one oval.*

- ○ Sim
- ○ Não

10. **Você costuma remover anomalias de código dos projetos que você participa?**

*Mark only one oval.*

- ○ Sim
- ○ Não
- ○ Não se aplica

## Refatorações

11. **Você tem conhecimento sobre refatorações?** *

*Mark only one oval.*

- ○ Sim
- ○ Não

12. **Você costuma refatorar os projetos que você participa?**

*Mark only one oval.*

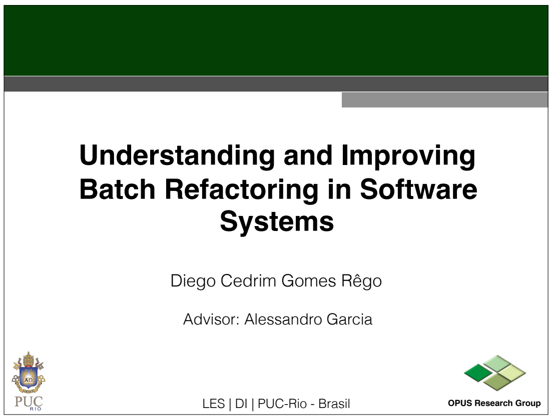
- ○ Sim
- ○ Não
- ○ Não se aplica

## Agradecemos sua colaboração!

Powered by Google Forms

# C
# Presentation

Following we present the slides that we used during the thesis presentation session.



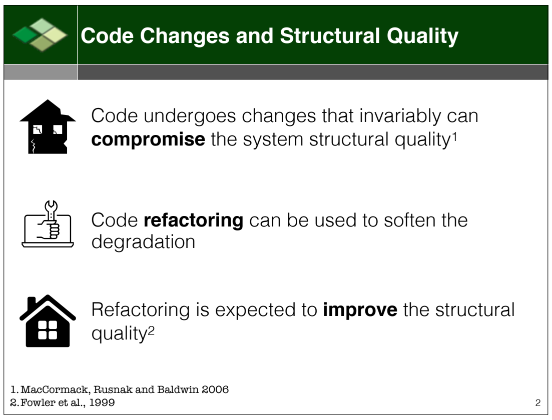
## Understanding and Improving Batch Refactoring in Software Systems

Diego Cedrim Gomes Rêgo

Advisor: Alessandro Garcia

LES | DI | PUC-Rio - Brasil

OPUS Research Group



## Code Changes and Structural Quality

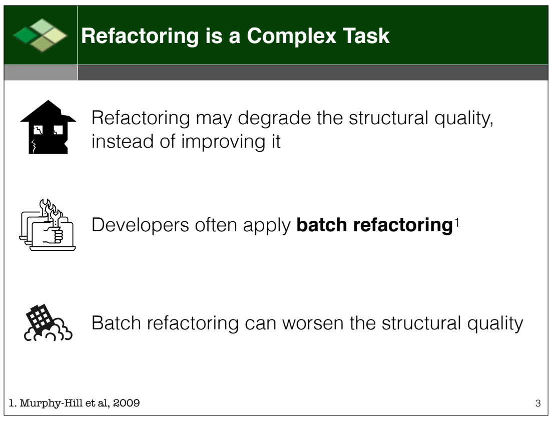
Code undergoes changes that invariably can **compromise** the system structural quality[1]

Code **refactoring** can be used to soften the degradation

Refactoring is expected to **improve** the structural quality[2]

1. MacCormack, Rusnak and Baldwin 2006
2. Fowler et al., 1999

2



## Refactoring is a Complex Task

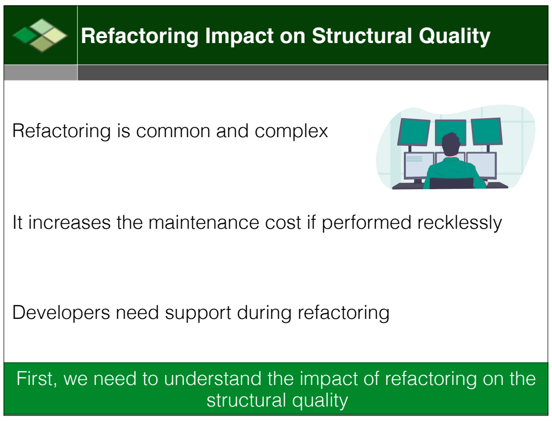
Refactoring may degrade the structural quality, instead of improving it

Developers often apply **batch refactoring**[1]

Batch refactoring can worsen the structural quality

1. Murphy-Hill et al, 2009

3



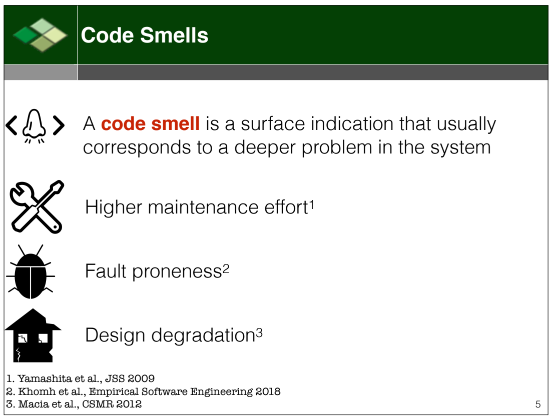
## Refactoring Impact on Structural Quality

Refactoring is common and complex

It increases the maintenance cost if performed recklessly

Developers need support during refactoring

First, we need to understand the impact of refactoring on the structural quality



## Code Smells

A **code smell** is a surface indication that usually corresponds to a deeper problem in the system

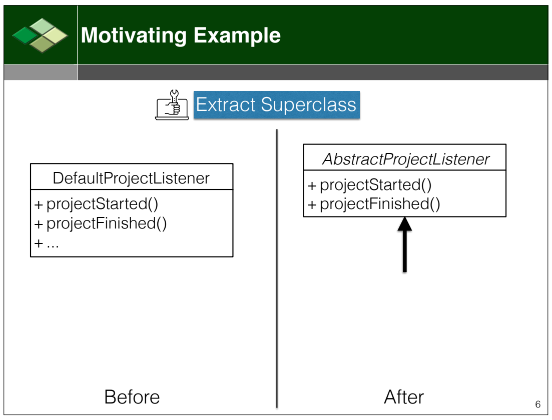
Higher maintenance effort[1]

Fault proneness[2]

Design degradation[3]

1. Yamashita et al., JSS 2009
2. Khomh et al., Empirical Software Engineering 2018
3. Macia et al., CSMR 2012

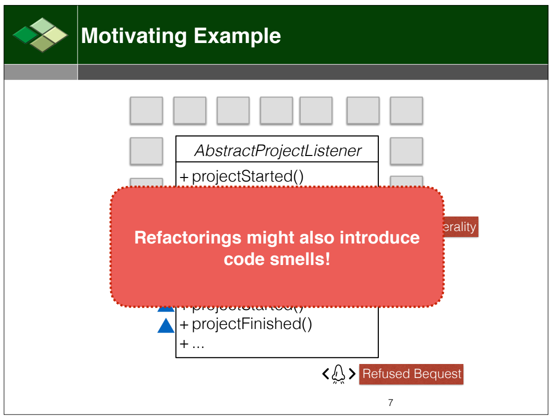
5



## Motivating Example

Extract Superclass

6



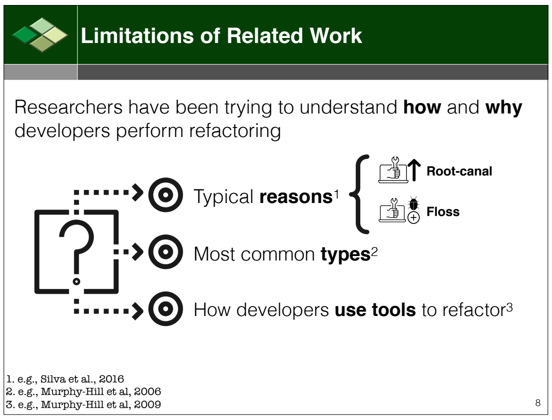
## Motivating Example

AbstractProjectListener
+ projectStarted()
+ projectFinished()
+ ...

Refactorings might also introduce code smells!

erality
Refused Bequest

7



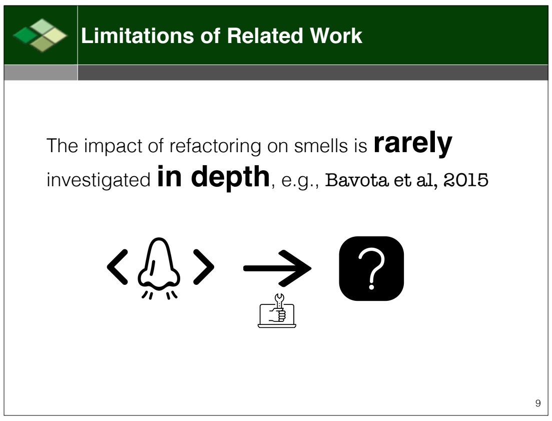
## Limitations of Related Work

Researchers have been trying to understand **how** and **why** developers perform refactoring

- Typical **reasons**[1]
  - Root-canal
  - Floss
- Most common **types**[2]
- How developers **use tools** to refactor[3]

1. e.g., Silva et al., 2016
2. e.g., Murphy-Hill et al, 2006
3. e.g., Murphy-Hill et al, 2009

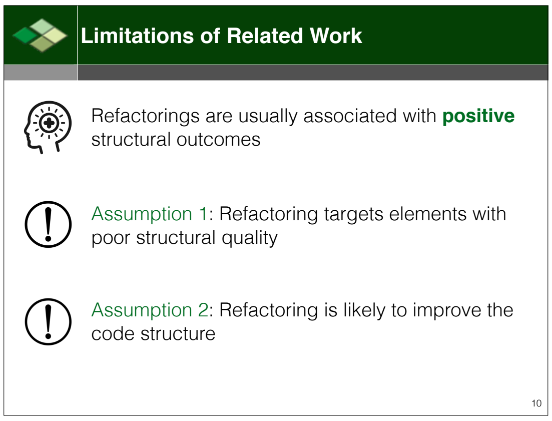
8



## Limitations of Related Work

The impact of refactoring on smells is **rarely** investigated **in depth**, e.g., Bavota et al, 2015

9



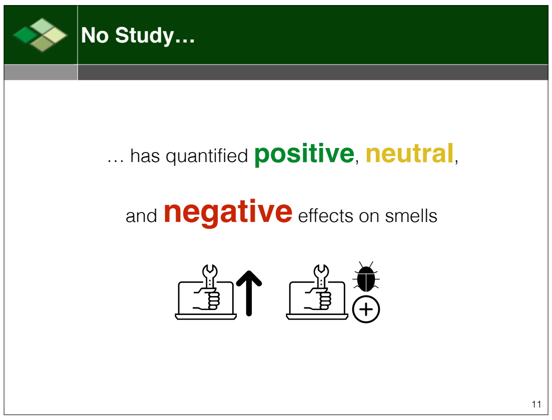
## Limitations of Related Work

Refactorings are usually associated with **positive** structural outcomes

Assumption 1: Refactoring targets elements with poor structural quality

Assumption 2: Refactoring is likely to improve the code structure

10

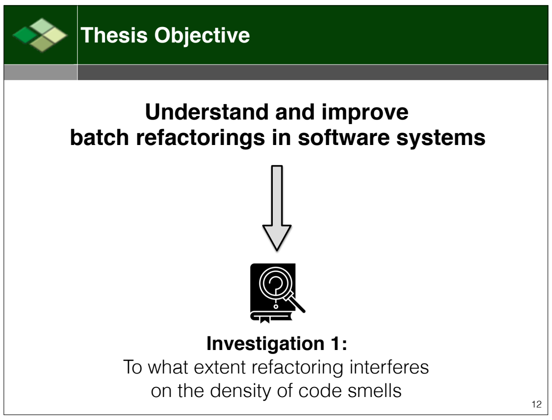


## No Study…

… has quantified **positive**, **neutral**,

and **negative** effects on smells

11

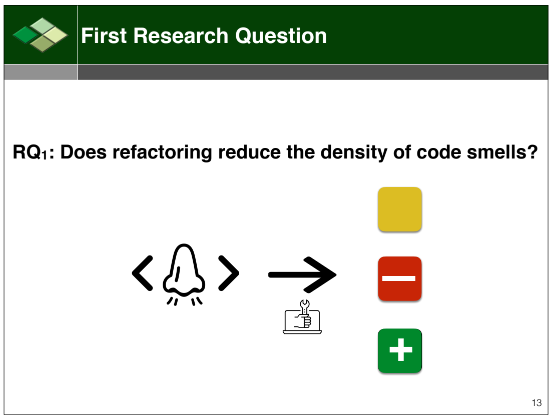


## Thesis Objective

**Understand and improve**
**batch refactorings in software systems**

**Investigation 1:**
To what extent refactoring interferes on the density of code smells

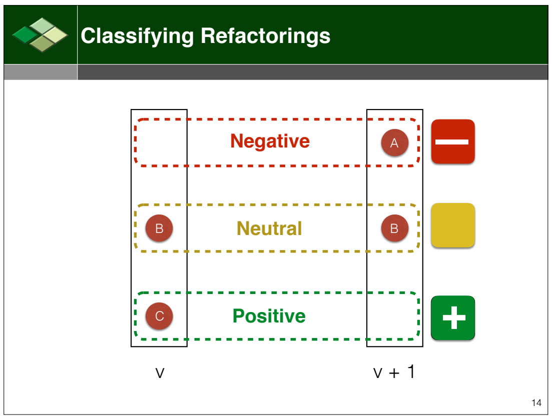
12



## First Research Question

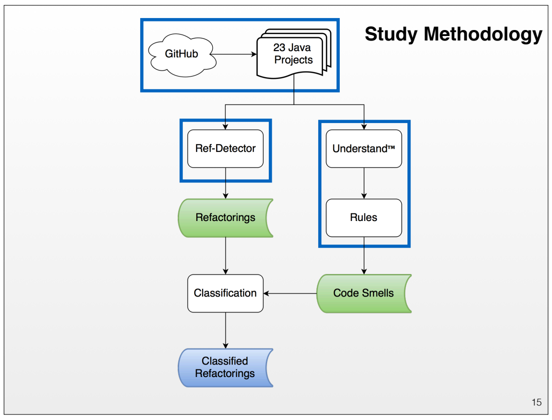
**RQ$_1$: Does refactoring reduce the density of code smells?**

13



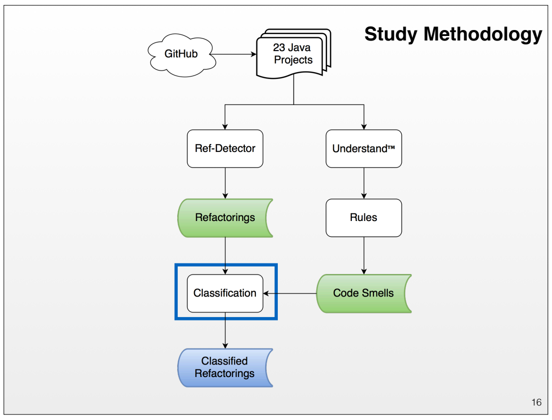
## Classifying Refactorings

14



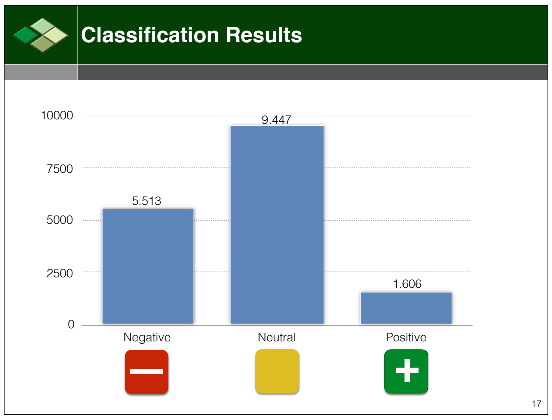
## Study Methodology

15



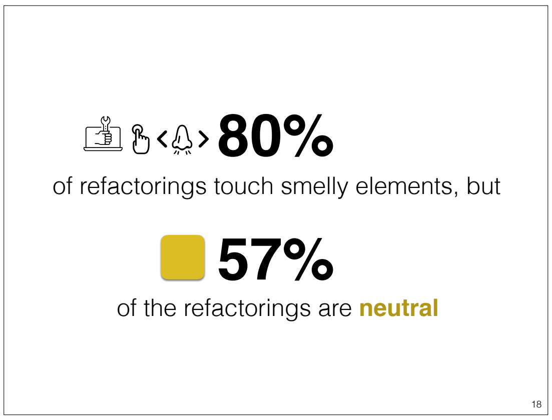
## Study Methodology

16



## Classification Results

17



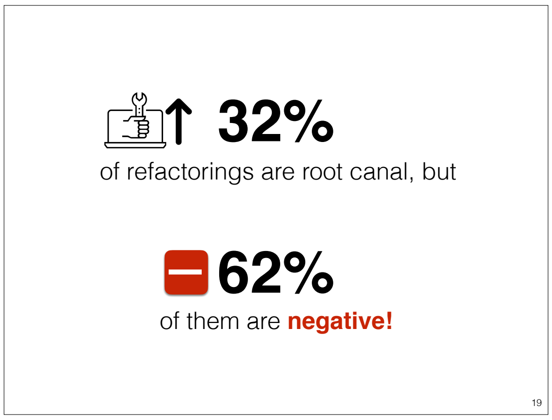
**80%**

of refactorings touch smelly elements, but

**57%**

of the refactorings are **neutral**

18



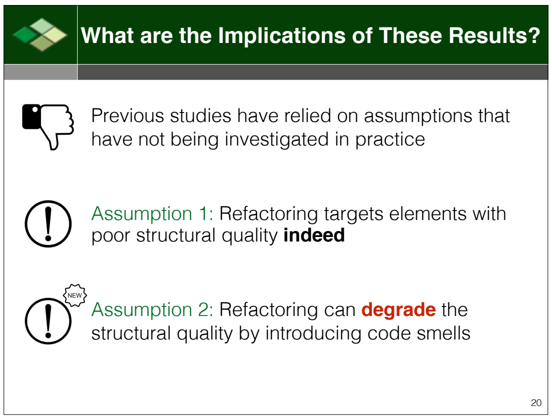
↑ 32%

of refactorings are root canal, but

62%

of them are **negative!**

19



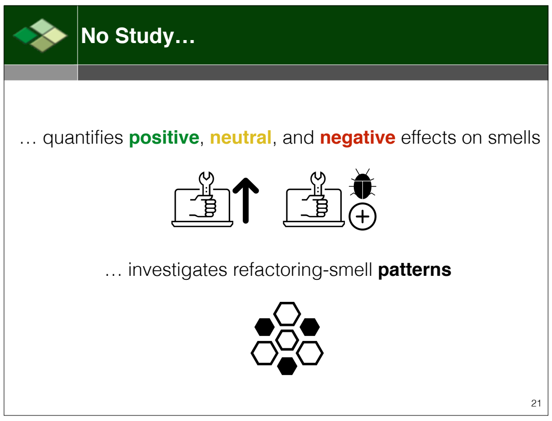
## What are the Implications of These Results?

Previous studies have relied on assumptions that have not being investigated in practice

Assumption 1: Refactoring targets elements with poor structural quality **indeed**

Assumption 2: Refactoring can **degrade** the structural quality by introducing code smells

20

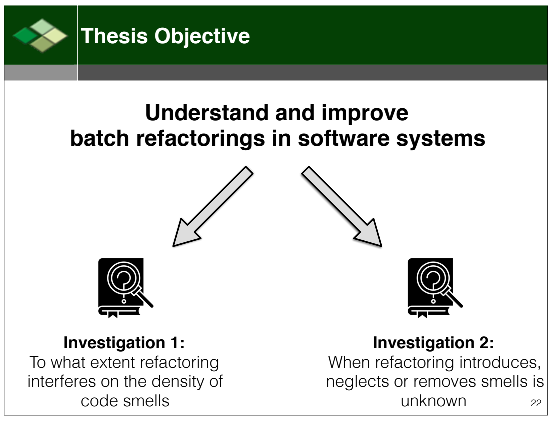


## No Study…

… quantifies **positive**, **neutral**, and **negative** effects on smells

… investigates refactoring-smell **patterns**

21



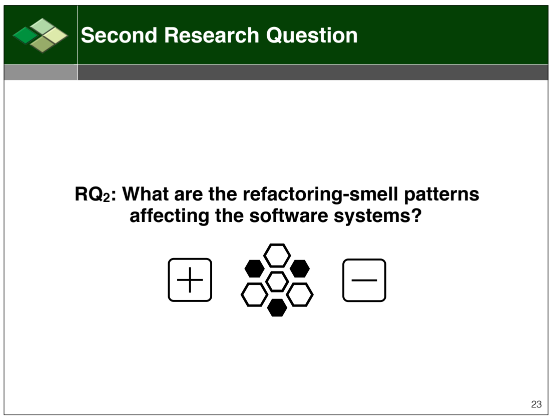
## Thesis Objective

**Understand and improve batch refactorings in software systems**

**Investigation 1:** To what extent refactoring interferes on the density of code smells

**Investigation 2:** When refactoring introduces, neglects or removes smells is unknown

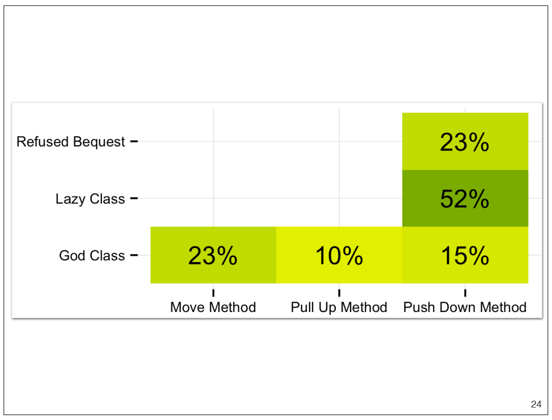
22



## Second Research Question

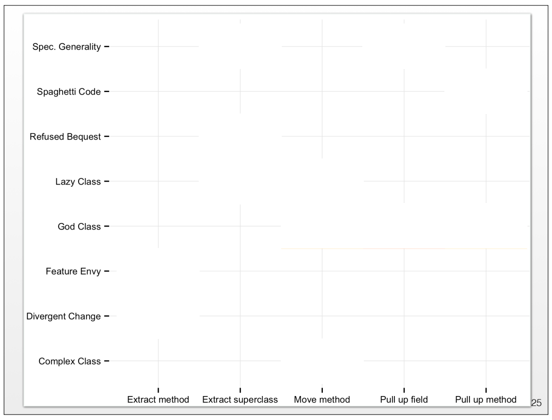
**RQ$_2$: What are the refactoring-smell patterns affecting the software systems?**

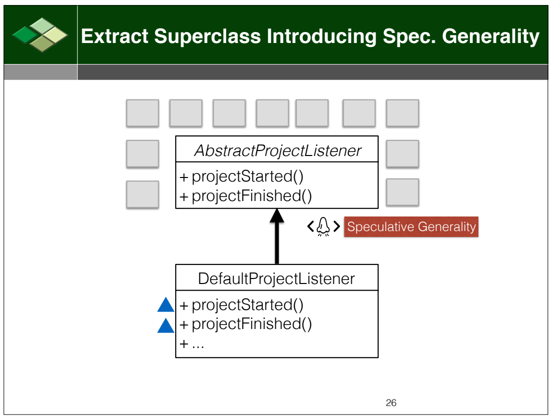
23



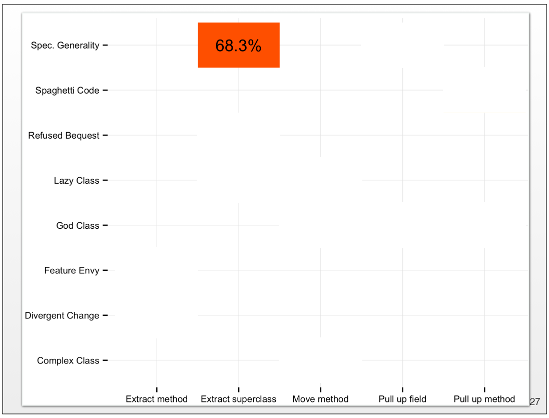
| | Move Method | Pull Up Method | Push Down Method |
|---|---|---|---|
| Refused Bequest | | | 23% |
| Lazy Class | | | 52% |
| God Class | 23% | 10% | 15% |

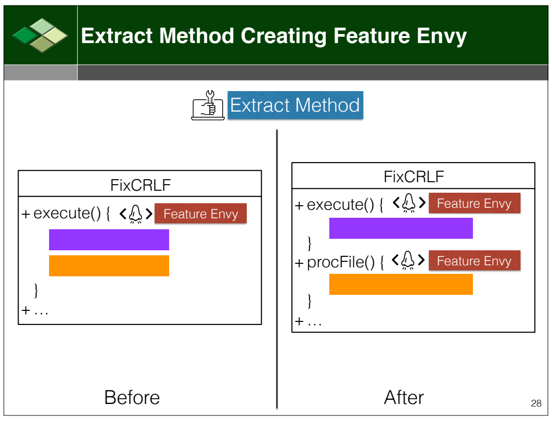
24





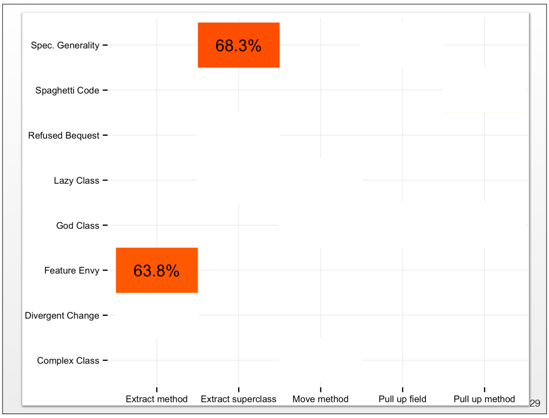
## Extract Superclass Introducing Spec. Generality

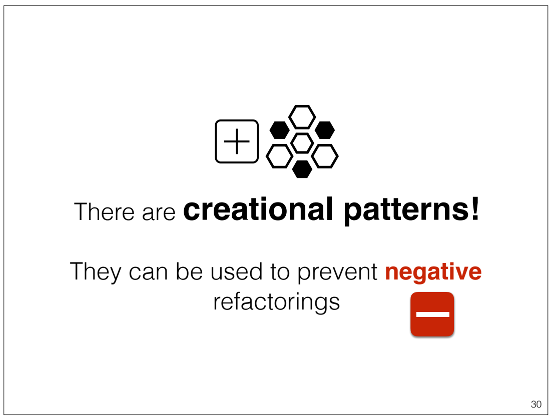




## Extract Method Creating Feature Envy

Extract Method
FixCRLF
+ execute() { Feature Envy
}
+ ...
Before
FixCRLF
+ execute() { Feature Envy
}
+ procFile() { Feature Envy
}
+ ...
After
28

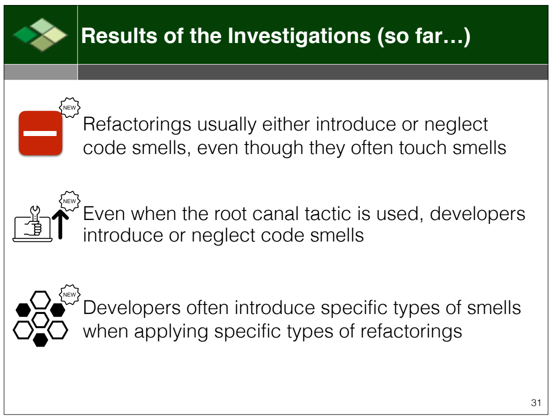




There are **creational patterns!**

They can be used to prevent **negative** refactorings

30



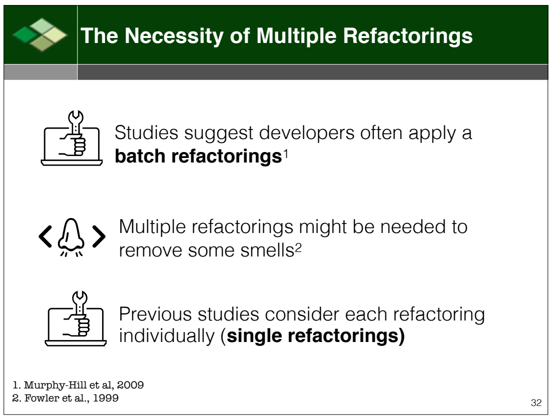
## Results of the Investigations (so far…)

Refactorings usually either introduce or neglect code smells, even though they often touch smells

Even when the root canal tactic is used, developers introduce or neglect code smells

Developers often introduce specific types of smells when applying specific types of refactorings

31



## The Necessity of Multiple Refactorings

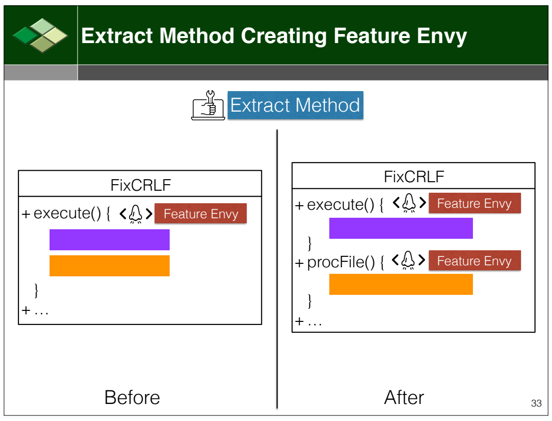
Studies suggest developers often apply a **batch refactorings**[1]

Multiple refactorings might be needed to remove some smells[2]

Previous studies consider each refactoring individually (**single refactorings)**

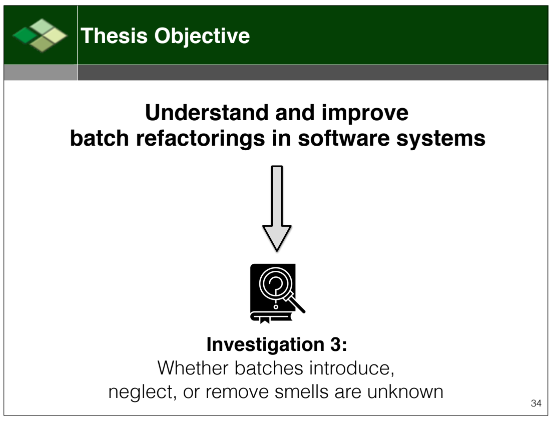
1. Murphy-Hill et al, 2009
2. Fowler et al., 1999

32



## Extract Method Creating Feature Envy

33

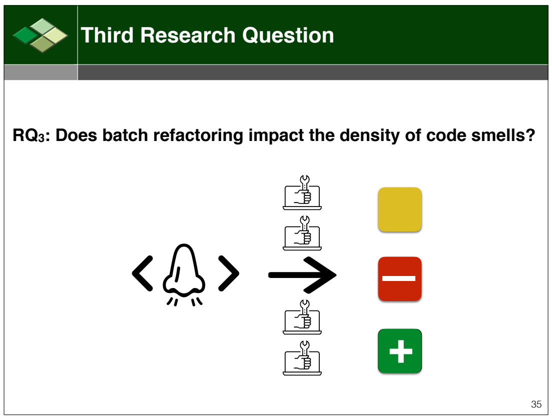


## Thesis Objective

**Understand and improve batch refactorings in software systems**

**Investigation 3:**
Whether batches introduce, neglect, or remove smells are unknown

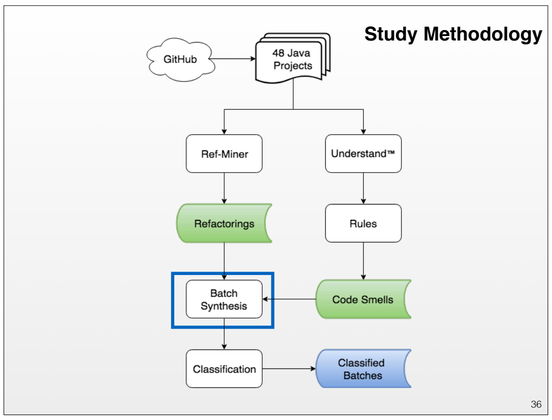
34



## Third Research Question

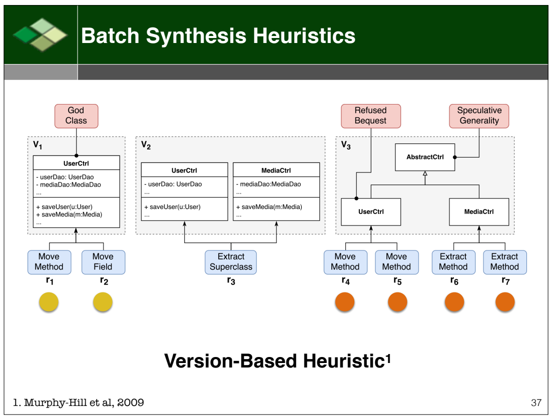
**RQ3: Does batch refactoring impact the density of code smells?**

35



## Study Methodology

36

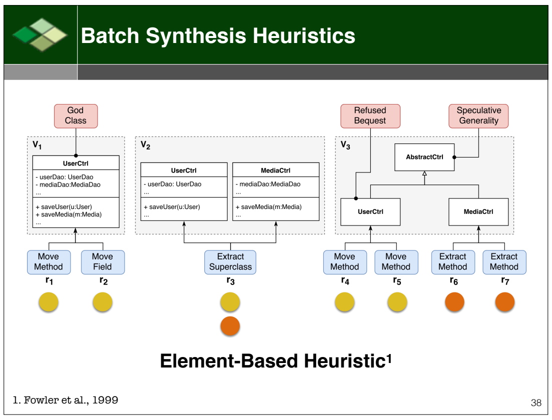


## Batch Synthesis Heuristics

**Version-Based Heuristic[1]**

1. Murphy-Hill et al, 2009

37

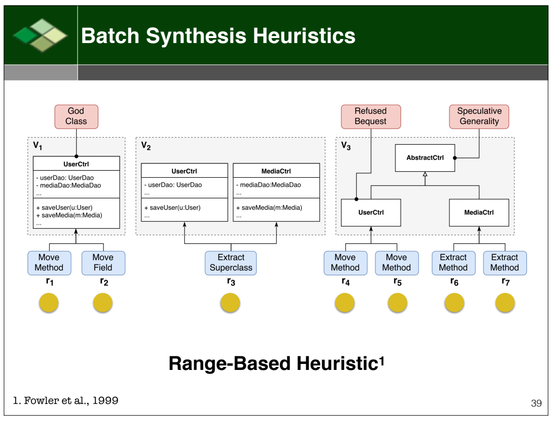


## Batch Synthesis Heuristics

**Element-Based Heuristic[1]**

1. Fowler et al., 1999

38

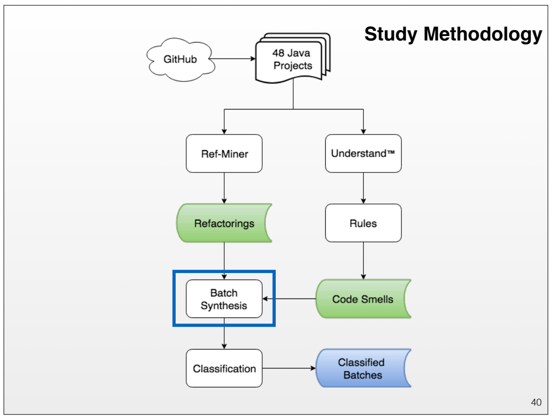


## Batch Synthesis Heuristics

**Range-Based Heuristic[1]**

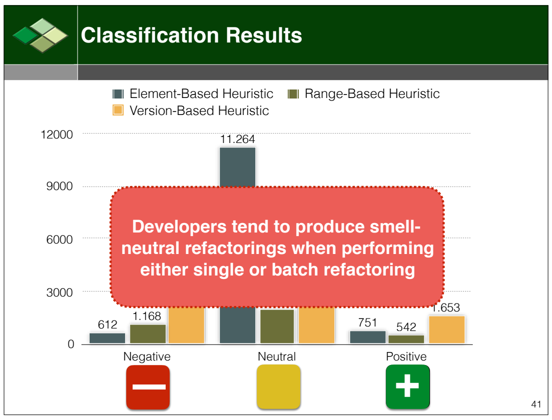
1. Fowler et al., 1999

39



## Study Methodology

40



## Classification Results

**Developers tend to produce smell-neutral refactorings when performing either single or batch refactoring**

41

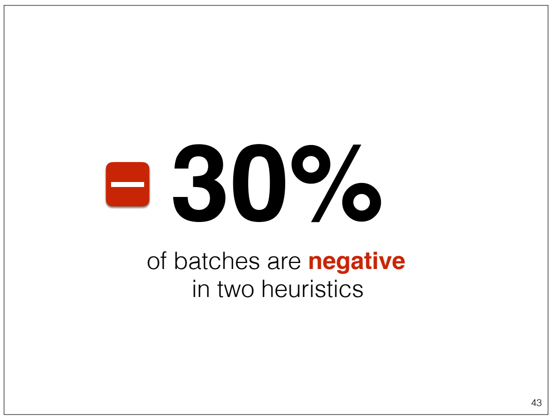
**14%**

of the batches are **positive**, while

**9.7%**

of the single refactorings were

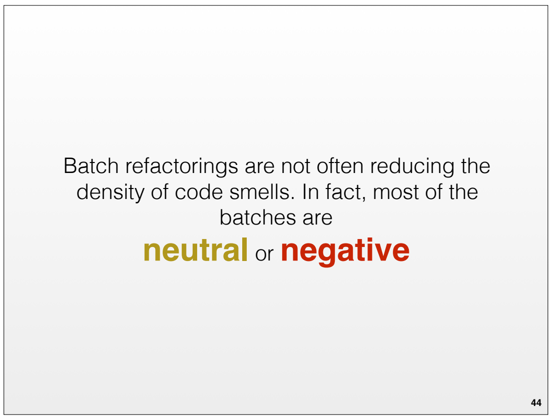
42



30%

of batches are **negative** in two heuristics

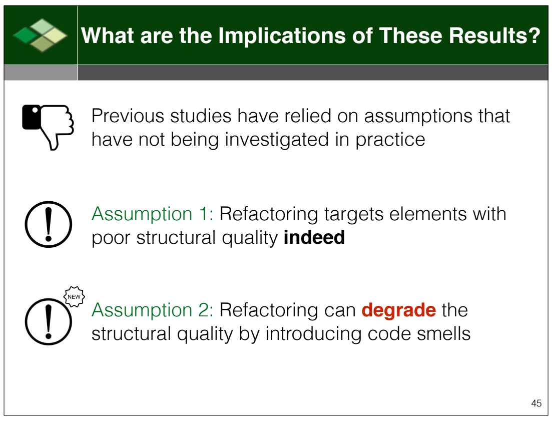
43



Batch refactorings are not often reducing the density of code smells. In fact, most of the batches are

**neutral** or **negative**

44



## What are the Implications of These Results?

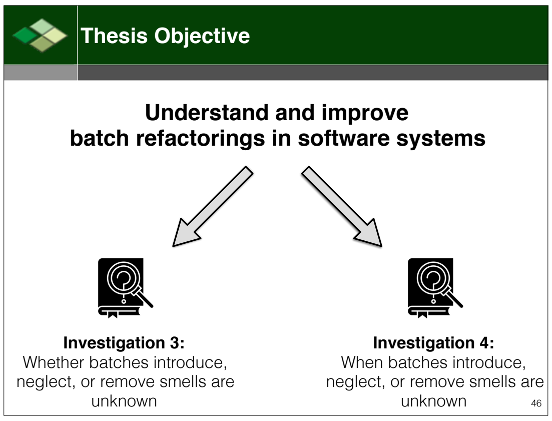
Previous studies have relied on assumptions that have not being investigated in practice

Assumption 1: Refactoring targets elements with poor structural quality **indeed**

NEW

Assumption 2: Refactoring can **degrade** the structural quality by introducing code smells

45



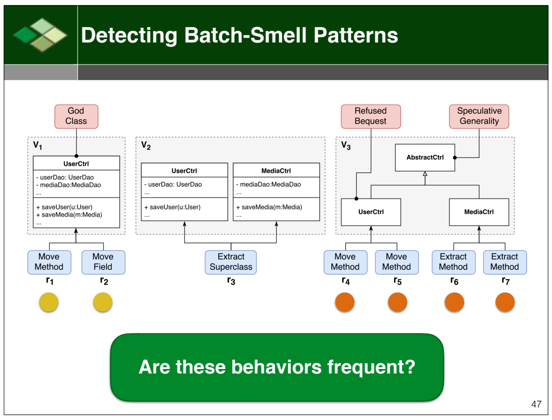
## Thesis Objective

**Understand and improve batch refactorings in software systems**

**Investigation 3:**
Whether batches introduce, neglect, or remove smells are unknown

**Investigation 4:**
When batches introduce, neglect, or remove smells are unknown

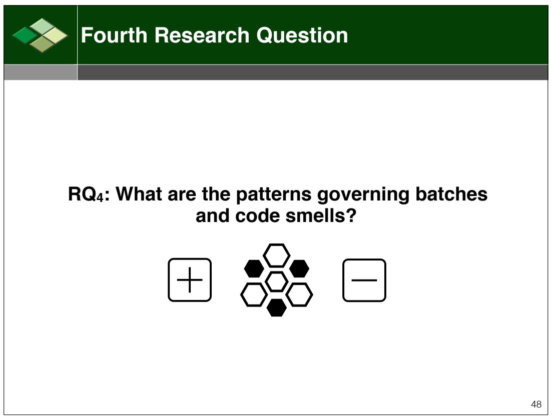
46



## Detecting Batch-Smell Patterns

**Are these behaviors frequent?**

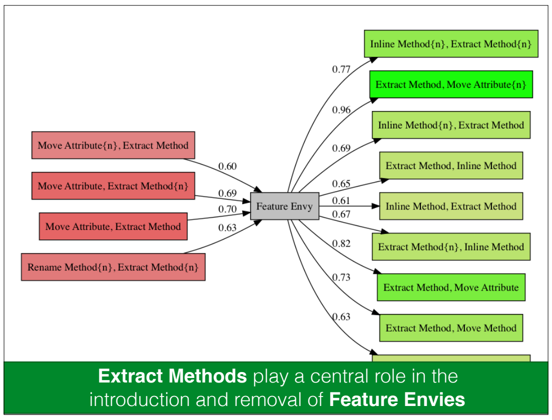
47



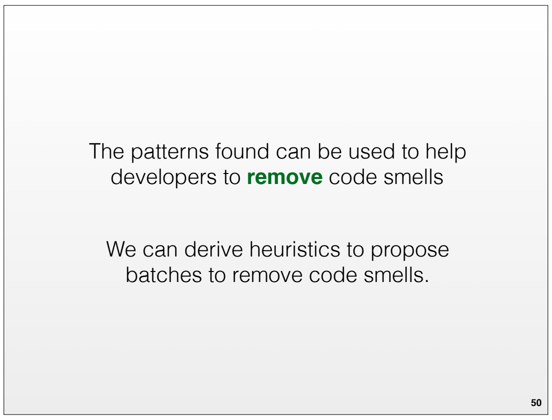
## Fourth Research Question

**RQ4: What are the patterns governing batches and code smells?**

48


Move Attribute{n}, Extract Method
Move Attribute, Extract Method{n}
Move Attribute, Extract Method
Rename Method{n}, Extract Method{n}
0.60
0.69
0.70
0.63
Feature Envy
0.77
0.96
0.69
0.65
0.61
0.67
0.82
0.73
0.63
Inline Method{n}, Extract Method{n}
Extract Method, Move Attribute{n}
Inline Method{n}, Extract Method
Extract Method, Inline Method
Inline Method, Extract Method
Extract Method{n}, Inline Method
Extract Method, Move Attribute
Extract Method, Move Method

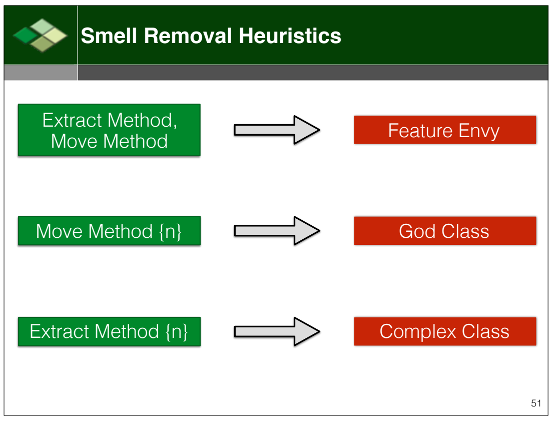
**Extract Methods** play a central role in the introduction and removal of **Feature Envies**



The patterns found can be used to help developers to **remove** code smells

We can derive heuristics to propose batches to remove code smells.

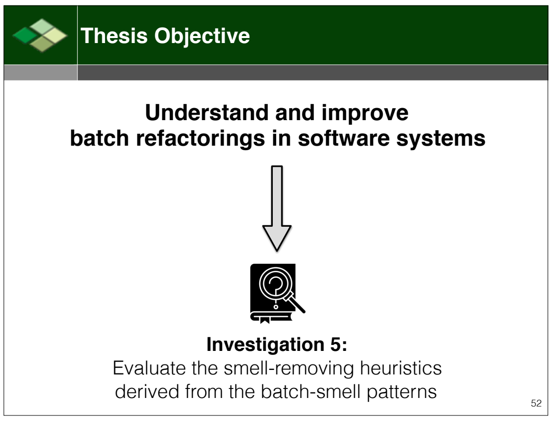
50



## Smell Removal Heuristics

| | |
|---|---|
| Extract Method, Move Method | Feature Envy |
| Move Method {n} | God Class |
| Extract Method {n} | Complex Class |

51

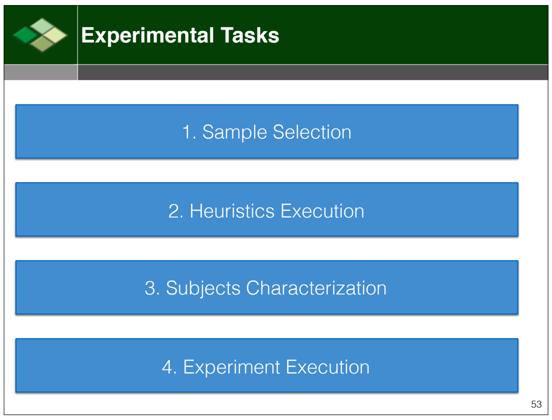


## Thesis Objective

**Understand and improve batch refactorings in software systems**

**Investigation 5:**
Evaluate the smell-removing heuristics derived from the batch-smell patterns

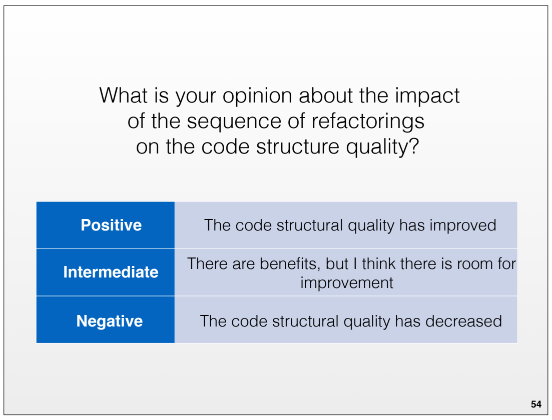
52



## Experimental Tasks

1. Sample Selection
2. Heuristics Execution
3. Subjects Characterization
4. Experiment Execution

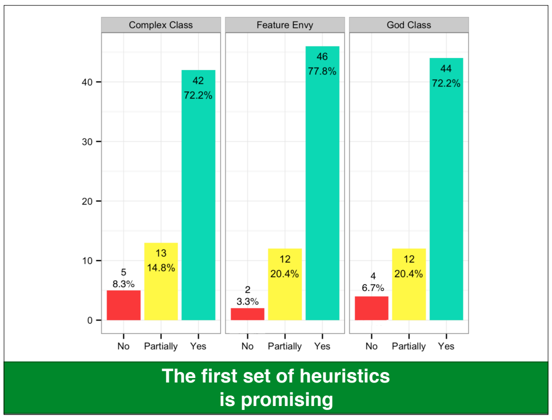
53



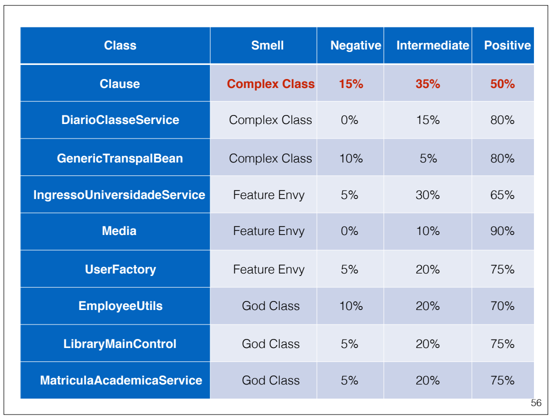
What is your opinion about the impact of the sequence of refactorings on the code structure quality?

| | |
|---|---|
| **Positive** | The code structural quality has improved |
| **Intermediate** | There are benefits, but I think there is room for improvement |
| **Negative** | The code structural quality has decreased |

54

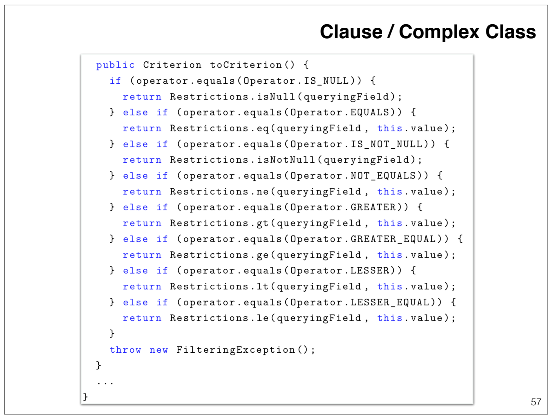


**The first set of heuristics is promising**



| Class | Smell | Negative | Intermediate | Positive |
|---|---|---|---|---|
| **Clause** | **Complex Class** | **15%** | **35%** | **50%** |
| **DiarioClasseService** | Complex Class | 0% | 15% | 80% |
| **GenericTranspalBean** | Complex Class | 10% | 5% | 80% |
| **IngressoUniversidadeService** | Feature Envy | 5% | 30% | 65% |
| **Media** | Feature Envy | 0% | 10% | 90% |
| **UserFactory** | Feature Envy | 5% | 20% | 75% |
| **EmployeeUtils** | God Class | 10% | 20% | 70% |
| **LibraryMainControl** | God Class | 5% | 20% | 75% |
| **MatriculaAcademicaService** | God Class | 5% | 20% | 75% |

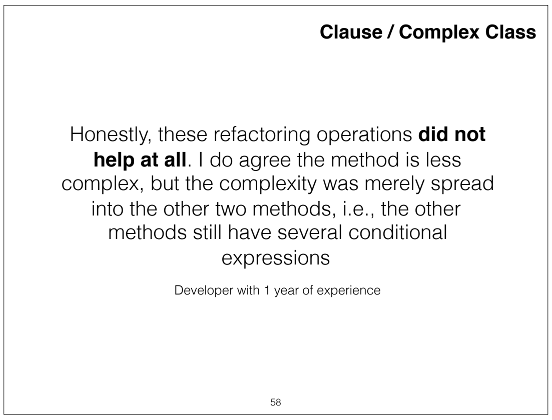
56



## Clause / Complex Class

```
public Criterion toCriterion() {
  if (operator.equals(Operator.IS_NULL)) {
    return Restrictions.isNull(queryingField);
  } else if (operator.equals(Operator.EQUALS)) {
    return Restrictions.eq(queryingField, this.value);
  } else if (operator.equals(Operator.IS_NOT_NULL)) {
    return Restrictions.isNotNull(queryingField);
  } else if (operator.equals(Operator.NOT_EQUALS)) {
    return Restrictions.ne(queryingField, this.value);
  } else if (operator.equals(Operator.GREATER)) {
    return Restrictions.gt(queryingField, this.value);
  } else if (operator.equals(Operator.GREATER_EQUAL)) {
    return Restrictions.ge(queryingField, this.value);
  } else if (operator.equals(Operator.LESSER)) {
    return Restrictions.lt(queryingField, this.value);
  } else if (operator.equals(Operator.LESSER_EQUAL)) {
    return Restrictions.le(queryingField, this.value);
  }
  throw new FilteringException();
}
...
}
```

57

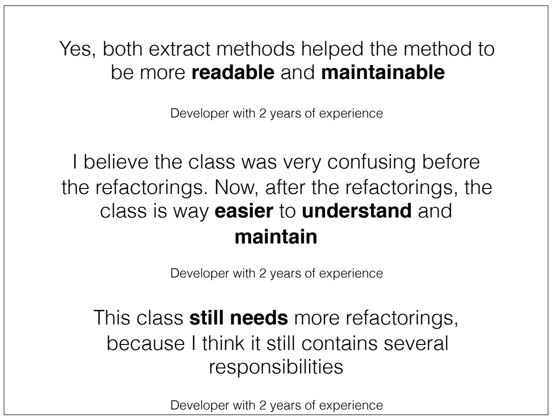


## Clause / Complex Class

Honestly, these refactoring operations **did not help at all**. I do agree the method is less complex, but the complexity was merely spread into the other two methods, i.e., the other methods still have several conditional expressions

Developer with 1 year of experience

58



Yes, both extract methods helped the method to be more **readable** and **maintainable**

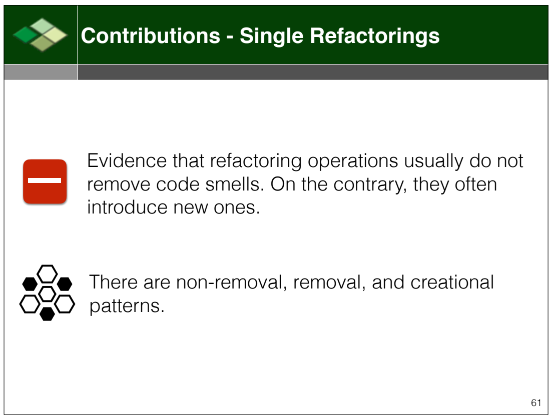
Developer with 2 years of experience

I believe the class was very confusing before the refactorings. Now, after the refactorings, the class is way **easier** to **understand** and **maintain**

Developer with 2 years of experience

This class **still needs** more refactorings, because I think it still contains several responsibilities

Developer with 2 years of experience

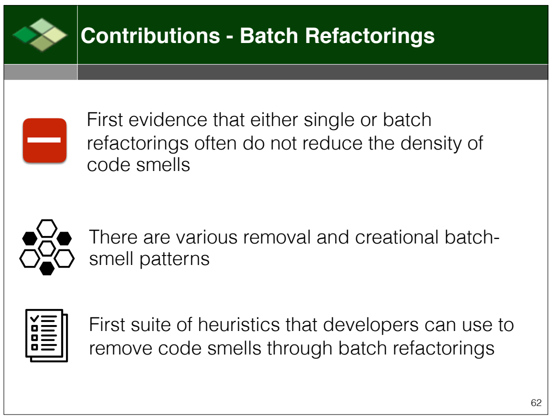


## Contributions - Single Refactorings

Evidence that refactoring operations usually do not remove code smells. On the contrary, they often introduce new ones.

There are non-removal, removal, and creational patterns.

61



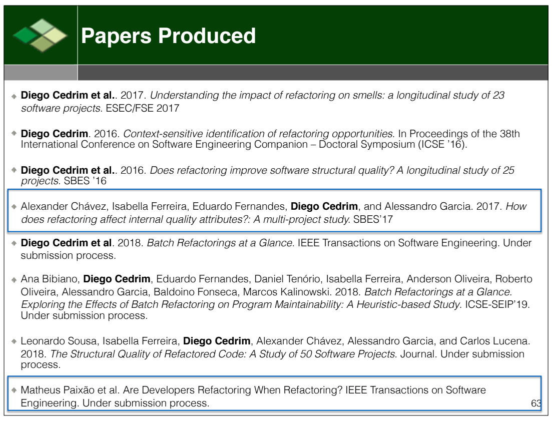
## Contributions - Batch Refactorings

First evidence that either single or batch refactorings often do not reduce the density of code smells

There are various removal and creational batch-smell patterns

First suite of heuristics that developers can use to remove code smells through batch refactorings

62



## Papers Produced

- **Diego Cedrim et al.**. 2017. *Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects.* ESEC/FSE 2017
- **Diego Cedrim**. 2016. *Context-sensitive identification of refactoring opportunities.* In Proceedings of the 38th International Conference on Software Engineering Companion – Doctoral Symposium (ICSE '16).
- **Diego Cedrim et al.**. 2016. *Does refactoring improve software structural quality? A longitudinal study of 25 projects*. SBES '16
- Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, **Diego Cedrim**, and Alessandro Garcia. 2017. *How does refactoring affect internal quality attributes?: A multi-project study.* SBES'17
- **Diego Cedrim et al**. 2018. *Batch Refactorings at a Glance*. IEEE Transactions on Software Engineering. Under submission process.
- Ana Bibiano, **Diego Cedrim**, Eduardo Fernandes, Daniel Tenório, Isabella Ferreira, Anderson Oliveira, Roberto Oliveira, Alessandro Garcia, Baldoino Fonseca, Marcos Kalinowski. 2018. *Batch Refactorings at a Glance. Exploring the Effects of Batch Refactoring on Program Maintainability: A Heuristic-based Study*. ICSE-SEIP'19. Under submission process.
- Leonardo Sousa, Isabella Ferreira, **Diego Cedrim**, Alexander Chávez, Alessandro Garcia, and Carlos Lucena. 2018. *The Structural Quality of Refactored Code: A Study of 50 Software Projects*. Journal. Under submission process.
- Matheus Paixão et al. Are Developers Refactoring When Refactoring? IEEE Transactions on Software Engineering. Under submission process.

63